ACCELERATED DEEP LEARNING FOR THE EDGE-TO-CLOUD CONTINUUM: A SPECIALIZED FULL STACK DERIVED FROM ALGORITHMS

A Dissertation
Presented to
The Academic Faculty

By

Hardik Sharma

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

May 2019

ACCELERATED DEEP LEARNING FOR THE EDGE-TO-CLOUD CONTINUUM: A SPECIALIZED FULL STACK DERIVED FROM ALGORITHMS

Approved by:

Dr. Hadi Esmaeilzadeh, Advisor
School of Computer Science and Engineering
University of California, San Diego

Dr. Hyesoon Kim
School of Computer Science,
College of Computing
Georgia Institute of Technology

Dr. Milos Prvulovic
School of Computer Science,
College of Computing
Georgia Institute of Technology

Dr. Tushar Krishna
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Vikas Chandra
AI Research
Facebook

Date Approved: March 15, 2019

Do not go where the path may lead, go instead where there is no path and leave a trail

Robert Frost

Dedicated to my family - Papa, Mummy, Chikki didi, and especially to you, Payal.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Dr. Hadi Esmaeilzadeh. Dr. Hadi has opened up countless opportunities for research and collaboration during the course of my PhD and continues to be a pillar of support beyond my graduation. He has taught me when to aim for perfection and when to just get-things-done. For the lessons and experience he has hammered into me, I am forever in his debt.

I thank my committee members, Dr. Hyesoon Kim, Dr. Milos Prvulovic, Dr. Tushar Krishna, and Dr. Vikas Chandra for their insightful discussions and invaluable feedback.

I had the privilege of learning from Dr. Naveen Suda, Dr. Liangzhen Lai, and Dr. Vikas Chandra during my internships at ARM's ML&IoT group. Working with ARM's ML&IoT group gave me unparalleled exposure to research opportunities and challenges. I am grateful for their countless hours of discussions and immense help with formulating ideas and writing papers.

I would like to thank Dr. Tom Olson, Alpana Kaulgud, and especially Alex Chalfin who gave me the opportunity to work with some of the best minds in the industry through an internship at ARM's GPU architecture and research group. Working in ARM's GPU group gave me a refreshed perspective into research and exposed me to the challenges and problems relevant to the industry. A special thanks to Alex Chalfin for continuing to mentor me throughout my career.

I am incredibly lucky to have Jongse Park, Amir Yazdanbaksh, Jacob Sacks, Divya Mahajan, and Michael Isaev as my colleagues at Georgia Tech. I thank my dear colleagues for their constant guidance that helped me keep pushing throughout my PhD journey. I will forever cherish the memories of our invigorating, fun, and sometimes embarrassing, breaks. I have learned a lot from all of you, in both my personal life and academic life.

I would like to thank Dr. Indrani Kar who advised my thesis during my undergraduate education at the Indian Institute of Technology, Guwahati. She instilled in me a curiosity

and hunger for knowledge that I will carry with me forever.

I wholeheartedly thank my father Dr. Ashok Kumar Sharma, my mother Dr. Pushplata Sharma, my sister Dr. Niharika Sharma, and my brother-in-law Dr. Rajat Maheshwari. I thank them for being the best role models I will always look up to. I also thank my father-in-law Dr. Ranjeet Singh Bagga, my mother-in-law Mrs. Sweety Bagga, and my brother-in-law Mr. Prabh Preet Bagga for their love and support. I am lucky to have a wonderful family that has always supported me throughout my education and career.

To my wife, Payal Bagga I am eternally grateful. Thank you for being the beacon of light. My PhD journey would not have been possible without your constant support, encouragement, and countless hours of proof-reading.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Advances in high-performance computer architecture design have been a major driver for the rapid evolution of Deep Neural Networks (DNN). Due to their insatiable demand for compute power, naturally, both the research community as well the industry have turned to accelerators to accommodate modern DNN computation. Furthermore, DNNs are gaining prevalence and have found applications across a wide spectrum of devices, from commodity smartphones to enterprise cloud platforms. However, there is no one-size-fits-all solution for this continuum of devices that can meet the strict energy/power/chip-area budgets for edge devices *and* meet the high performance requirements for enterprise-grade servers.

To this end, this thesis designs a specialized compute stack for DNN acceleration across the edge-to-cloud continuum that flexibly matches the varying constraints for different devices and simultaneously exploits algorithmic properties to maximize the benefits from acceleration. As such, this thesis is divided into four thrusts:

Thrust 1: Neural acceleration for GPU throughput processors. This thesis first explores a tight integration of Neural Network (NN) accelerators *within* the massively-parallel GPUs with a minimal area overhead. We show that a tight-coupling of NN-accelerators and GPUs can provide a significant gain in performance and energy efficiency across a diverse set of applications through neural acceleration, by approximating regions of approximation-amenable code using a neural networks. Chapter 1 explores this tight integration of neural accelerators within the highly parallel GPUs in detail.

Thrust 2: Specialized compute-stack for accelerating Deep Neural Networks using FPGAs. Next, this thesis develops a full-stack for accelerating DNN *inference* on FPGAs that aims to provide *programmability*, *performance*, and *efficiency*. We call our specialized compute stack DNNWEAVER, which encompasses (1) high-level algorithmic abstractions, (2) a flexible template accelerator architecture, and (3) a compiler that automatically and

efficiently optimizes the template architecture to maximize DNN performance using the limited resources available on the FPGA die. Chapter 2 discusses DNNWEAVER in detail.

Thrust 3: Scale-out acceleration for training statistical machine learning. The third thrust of this thesis explores scale-out acceleration of *training* using cloud-scale FPGAs for a wide range of machine learning algorithms, including neural networks. The challenge here is to design an accelerator architecture that can scale up to efficiently use the large pool of compute resources available on modern cloud-grade FPGAs. To tackle this challenge, this thesis explores multi-threading to maximize efficiency from FPGA acceleration by running multiple parallel threads of training. Chapter 3 discusses the system design for scale-out acceleration in detail.

Thrust 4: Leverage robustness to reduction in bitwidth for deep learning. The final thrust of this thesis builds upon the algorithmic insight that bitwidth of operations in DNNs can be reduced without compromising their classification accuracy. However, to prevent loss of accuracy, the bitwidth varies significantly across DNNs and it may even be adjusted for each layer individually. Thus, a fixed-bitwidth accelerator would either offer limited benefits to accommodate the worst-case bitwidth requirements, or inevitably lead to a degradation in final accuracy. To alleviate these deficiencies, the final thrust of this thesis introduces dynamic bit-level fusion/decomposition as a new dimension in the design of DNN accelerators. Chapter 4 first describes the Bit Fusion architecture, the first bit-level dynamically composable architecture for accelerating DNNs using digital circuits.

The final thrust of this thesis explores mixed-signal acceleration to push accelerator efficiency to its limits. As such, the final thrust explores executing the low-bitwidth multiply-add operations prevalent in DNNs in the analog domain to gain significant efficiency benefits. Using low-bitwdith analog compute units enables us to overcome the limited range for information encoding, susceptibility to noise, and Analog to Digital (A/D) conversion overheads. Next, Chapter 4 (Section 4.2) details the design of a 3D-stacked mixed-signal accelerator architecture that can provide significant gains in efficiency over purely-digital

state-of-the-art 3D-stacked accelerator, without losing any classification accuracy.

CHAPTER 1

ACCELERATING NEURAL EXECUTION IN GPUS FOR APPROXIMATE EXECUTION OF GENERAL-PURPOSE PROGRAMS

This first thesis chapter explores the application of Multi-Layer Perceptrons (MLPs), a subset of deep learning, to accelerate general purpose programs in GPUs through approximate execution. Graphics Processing Units (GPUs) can accelerate diverse classes of applications – including recognition, gaming, data analytics, weather prediction, and multimedia – that are amenable to approximate execution. Leveraging this insight, this chapter designs a low-cost neural network accelerator architecture, called NGPU, that can be tightly integrated within the highly-parallel GPU SIMD cores for *approximate neural execution*. The neurally accelerated NGPU architecture imposes a nominal area overhead (less than 1%) for GPUs and enables significant performance and efficiency gains across a wide range of applications, while limiting quality loss to just 2.5%. This chapter is based on work presented in the MICRO 2015 [1], and is a result of collaboration with Amir Yazdanbaksh[1], Jongse Park[1], Pejman Lotfi-Kamran [2], and Hadi Esmaeilzadeh[3].

1.1 Introduction

The availability of programming models for GPUs and the advances in their microarchitecture have played a significant role in their widespread adoption. Many companies, such as Microsoft, Google, and Amazon use GPUs to accelerate their enterprise services. As GPUs play a major role in accelerating many classes of applications, improving their performance and efficiency is imperative to enable new capabilities and to cope with the ever-increasing rate of data generation. Many of the applications that benefit from GPUs are also amenable

[1]Georgia Institute of Technology
[2]Institute for Research in Fundamental Sciences
[3]University of California, San Diego

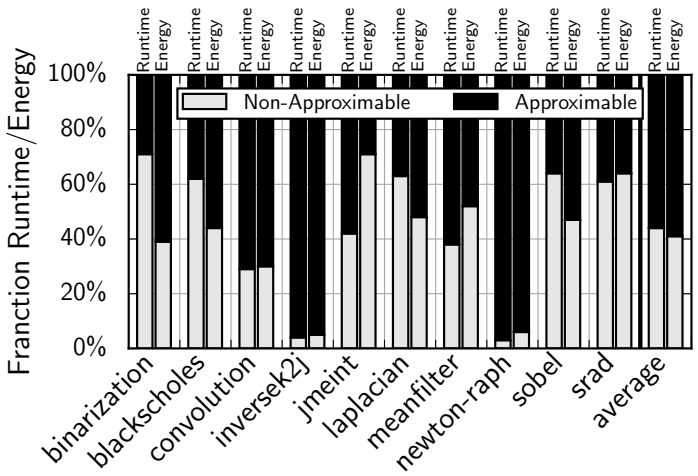


Figure 1.1: Runtime and energy breakdown between neurally approximable regions and the regions that cannot be approximated.

to imprecise computation [3, 4, 5, 6]. This characteristic of many GPU applications provides a unique opportunity to devise approximation techniques that trade small losses in the quality of results for significant gains in performance and efficiency.

Among approximation techniques, neural acceleration provides significant gains for CPUs [7, 8, 9, 10, 11] and may be a good candidate for GPUs. Neural acceleration relies on an automated algorithmic transformation that converts an approximable segment of code[4] to a neural network. This transformation is called the neural transformation [7]. The compiler automatically performs the neural transformation and replaces the approximable segment with an invocation of a neural hardware that accelerates the execution of that segment.

1.2 Motivation for neural acceleration in GPUs

To examine the potential benefits of neural acceleration in GPUs, we first study its applicability to a diverse set of representative CUDA applications. Figure 1.1 illustrates the results and shows the breakdown of application runtime and energy dissipation between

[4]Approximable code is a segment that if approximated will not lead to catastrophic failures in execution (e.g., segmentation fault) and its approximation may only lead to graceful degradation of the application output quality.

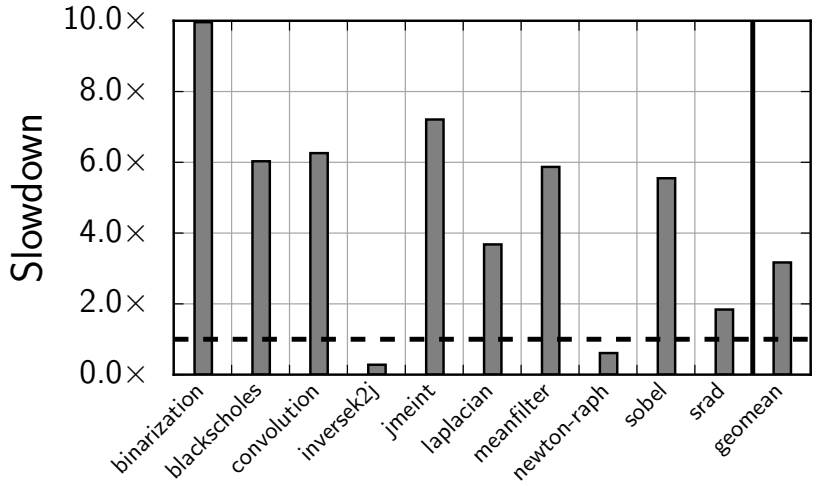


Figure 1.2: Slowdown with software-only neural transformation due to the lack of hardware support for neural acceleration.

neurally approximable regions and the regions that cannot be neurally approximated[5]. On average, applications spend 56% of their runtime and 59% of their energy in neurally approximable regions. Some applications such as inversek2j and newton-raph spend more than 93% of their runtime and energy in neurally approximable regions. These encouraging results demonstrate the significant potential of neural acceleration for GPU processors.

Why hardware acceleration? As previous work [12] suggested, it is possible to apply neural transformation with no hardware modifications and replace the approximable region with an efficient software implementation of the neural network that mimics the region. However, with a software-only solution, the applications suffer from 3.2× slowdown, as shown in in Figure 1.2. Only inversek2j and newton-raph, which spend more than 90% of their time in the neurally approximable region see performance benefits through a software-only solution. The slowdown with software implementation is due to (1) the overhead of fetching/decoding the instructions, (2) the cost of frequent accesses to the memory/register file, and (3) the overhead of executing the sigmoid function.

The significant potential of neural transformation (Figure 1.1) and the slowdown with the software-only approach necessitates designing GPU architectures with integrated neural

[5]The annotation procedure is out of scope of this thesis and is discussed in detail in our MICRO paper [1].

accelerators.

1.3 Contributions

To this end, the following are the major contributions of this thesis chapter.

- While this work is not the first to explore neural acceleration, it is the first to evaluate tight integration of neural acceleration within GPU cores. Integrating neural accelerators within GPUs is fundamentally different from doing so in a CPU because of the hardware constraints and the many-thread SIMT execution model in GPUs.
- We observe that, unlike CPUs, the added parallelism is not the main source of benefits from neural acceleration in GPUs. The gains in GPUs come from (1) eliminating the fetch/decode during neural execution, (2) reducing accesses to the memory/register file by storing the parameters and the partial results in small buffers within the SIMD lanes, and (3) implementing sigmoid as a lookup table. This insight leads to a low overhead integration of neural accelerators to SIMD lanes by limiting the number of ALUs in an accelerator to only the one that is already in a SIMD lane.

The following section describes the NGPU architecture in detail. The programming interface for NGPU, its compilation workflow, and the ISA extensions for GPU to enable NGPU are beyond the scope of this thesis and are discussed in detail in our MICRO paper [1].

1.4 NGPU Architecture

To describe our neural accelerator design and its integration into the GPU architecture, we assume a GPU processor based on the Nvidia Fermi. Fermi's SMs contain 32 double-clocked SIMD lanes that execute two half warps (16 threads) simultaneously, where each warp executes in lock-step. Ideally, to preserve the data-level parallelism across the threads and preserve the default SIMT execution model, each SM needs to be augmented with 32

neural accelerators. Therefore, the objective is to design a neural accelerator that can be replicated 32 times within each SM for a minimal hardware overhead. These two requirements fundamentally change the design space of the neural accelerator from prior work that aims at accelerating single-thread cores with only one accelerator.

A naïve approach is to replicate and add the previously proposed CPU neural accelerator to each SM [7]. These CPU specific accelerators harbor multiple processing engines and contain significant amount of buffering for weights and control. Such a design not only imposes significant hardware overhead, but also is an overkill for data-parallel GPU architectures as our MICRO paper shows [1]. Instead, NGPU tightly integrates a GPU specific neural network in every SIMD lane.

The neural algorithmic transformation uses Multilayer Perceptrons (MLPs) to approximate CUDA code segments. An MLP consists of a network of neurons arranged in multiple layers. Each neuron in a layer is connected to all of the neurons in the next layer. Each neuron input is associated with a weight value that is generated after training. All neurons are identical and each neuron computes its output (y) based on $y = sigmoid(\sum_i (w_i \times x_i))$, where x_i is a neuron input and w_i is the input's associated weight. Therefore, all the computations of a neural network are a set of multiply-add operations followed by the nonlinear sigmoid operation. The neural accelerator only needs to support these two operations.

1.4.1 Integrating the Neural Accelerator

Each SM has 32 SIMD lanes, divided into two 16-lane groups that execute two half warps simultaneously. The ALU in each lane supports floating point multiply-add operation. We reuse these ALUs while enhancing the lanes for neural computation. We leverage the existing SIMT execution model to minimize the hardware overhead for the weights and control. We refer to the resulting SIMD lanes as neurally enhanced SIMD lanes.

In Figure 1.3, the added hardware components are numbered and highlighted in gray. The first component is the `Weight FIFO` (**1.**) that is a circular buffer and stores the

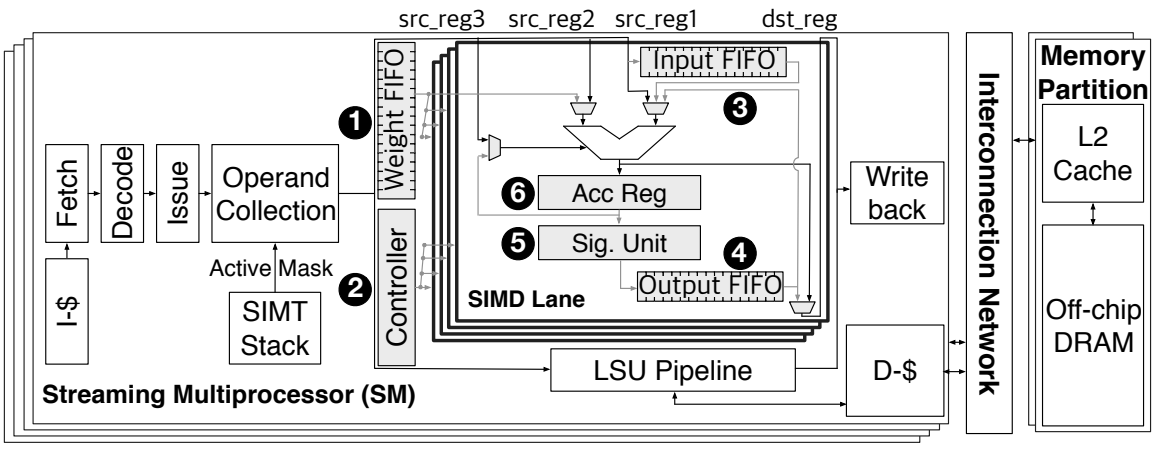


Figure 1.3: SM pipeline after integrating the neural accelerator within SIMD lanes. The added hardware is highlighted in gray.

synaptic weights. Since all of the threads are approximated by the same neural network, we only add one `Weight FIFO`, which is shared across all SIMD lanes. The Weight FIFO has two read ports corresponding to the two 16 SIMD lanes that execute two half warps. Each port supplies a weight to 16 ALUs. The second component is the `Controller` (**2.**) which controls the execution of the neural network across the SIMD lanes. Again, the `Controller` is shared across 16 SIMD lanes that execute a half warp (two controllers per SM). The `Controller` follows the SIMT pattern of execution for neural computation and enables the ALUs to perform the computation of the same input of the same neuron in the network.

We augment each of the SIMD lanes with an `Input FIFO` (**3.**) and an `Output FIFO` (**4.**). The `Input FIFO` stores the neural network inputs. The `Output FIFO` stores the output of the neurons including the output neurons that generate the final output. These two are small FIFO structures that are replicated for each SIMD lane. Each of the SIMD lanes also harbors a `Sigmoid Unit` (**5.**) that contains a read-only lookup table. This lookup table implements the nonlinear sigmoid function and is synthesized as combinational logic to reduce the area overhead. Finally, the `Acc Reg` (**6.**), which is the accumulator register in each of the SIMD lanes, retains the partial results of the sum of products ($\sum_i (w_i \times x_i)$) before passing it to the `Sigmoid Unit`.

Table 1.1: Applications, accelerated regions, training and evaluation datasets, quality metrics, and approximating neural networks.

	Description	Source	Domain	Quality Metric	# of Function Calls	# of Loops	# of ifs/ elses	# of PTX Insts.	Training Input Set	Evaluation Input Set	Digital NPU	
											Neural Network Topology	Quality Loss
binarization	Image binarization	Nvidia SDK	Image Processing	Image Diff	1	0	1	27	Three 512x512 pixel images	Twenty 2048x2048 pixel images	3 -> 4 -> 2 -> 1	8.23%
blackscholes	Option pricing	Nvidia SDK	Finance	Avg. Rel. Error	2	0	0	96	8,192 options	262,144 options	6 -> 8 -> 1	4.35%
convolution	Data filtering operation	Nvidia SDK	Machine Learning	Avg. Rel. Error	0	2	2	886	8,192 data points	262,144 data points	17 -> 2 -> 1	5.25%
inversek2j	Inverse kinematics for 2-joint arm	CUDA-Based Kinematics	Robotics	Avg. Rel. Error	0	3	5	132	8,192 2D coordinates	262,144 2D coordinates	2 -> 16 -> 3	8.73%
jmeint	Triangle intersection detection	jMonkey Game	3D Gaming	Miss Rate	4	0	37	2,250	8,192 3D coordinates	262,144 3D coordinates	18 -> 8 -> 2	17.32%
laplacian	Image sharpening filter	Nvidia SDK	Image Processing	Image Diff	0	2	1	51	Three 512x512 pixel images	Twenty 2048x2048 pixel images	9 -> 2 -> 1	6.01%
meanfilter	Image smoothing filter	Nvidia SDK	Machine Vision	Image Diff	0	2	1	35	Three 512x512 pixel images	Twenty 2048x2048 pixel images	7 -> 4 -> 1	7.06%
newton-raph	Newton-Raphson equation solver	Likelihood Estimators	Numerical Analysis	Avg. Rel. Error	2	2	1	44	8,192 cubic equations	262,144 cubic equations	5 -> 2 -> 1	3.08%
sobel	Edge detection	Nvidia SDK	Image Processing	Image Diff	0	2	1	86	Three 512x512 pixel images	Twenty 2048x2048 pixel images	9 -> 4 -> 1	5.45%
srad	Speckle reducing anisotropic diffusion	Rodinia	Medical Imaging	Image Diff	0	0	0	110	Three 512x512 pixel images	Twenty 2048x2048 pixel images	5 -> 4 -> 1	7.43%

One of the advantages of this design is that it limits all major modifications to the execution part of the SIMD lanes (pipelines). There is no need to change any other part of the SM except for adding support for decoding the ISA extensions that communicate data to the accelerator (i.e., input and output buffers). Scheduling and issuing these instructions are similar to arithmetic instructions and do not require specific changes.

1.5 Evaluation

We evaluate the benefits of the proposed architecture across different bandwidth and accelerator settings. We use a diverse set of applications, cycle-accurate simulation, logic synthesis, and consistent detailed energy modeling.

1.5.1 Applications and Neural Transformation

Applications. As Table 1.1 shows, we use a diverse set of *approximable* GPU applications from the Nvidia SDK [13] and Rodinia [14] benchmark suites to evaluate the integration of neural accelerators within GPU architectures. We added three more applications to the mix from different sources [15, 16, 17]. As shown, the benchmarks represent workloads from finance, machine learning, image processing, vision, medical imaging, robotics, 3D

gaming, and numerical analysis.

Annotations. We annotate the CUDA source code for each application using the #pragma directives. We use theses directives to delineate a region within a CUDA kernel that has fixed number of inputs/outputs and is safe to approximate. Although it is possible and may boost the benefits to annotate multiple regions, we only annotate one region that is easy to identify and is frequently executed. As illustrated by the numbers of function calls, conditionals, and loops in Table 1.1, these regions exhibit a rich and diverse control flow behavior.

Evaluation/training datasets. As illustrated in Table 1.1, the datasets that are used for measuring the quality, performance, and energy are completely disjoint from the ones used for training the neural networks. The training inputs are typical representative inputs (such as sample images) that can be found in application test suites. For instance, we use the image of lena, peppers, and mandrill for applications that operate on image data. Since the regions are frequently executed, even a single application input provides large number of training data. For example, in sobel a 512×512 pixel image generates 262,144 training data elements.

Neural networks. The Neural Network Topology column shows the topology of the neural network that replaces the region of code. For instance, the topology for blackscholes is 6 $\rightarrow$ 8 $\rightarrow$ 1. That is the neural network has 6 inputs, one hidden layer with 8 neurons, and 1 output neuron. These topologies are automatically discovered by our compiler and we use the 10-fold cross validation technique to train the neural networks. As the results suggest, different applications require different topologies. Therefore, the SM architecture should be changed in a way that is reconfigurable and can accommodate different topologies.

1.5.2 Experimental Setup

Cycle-accurate simulations. We use the GPGPU-Sim cycle-accurate simulator version 3.2.2 [18]. We modified the simulator to include our ISA extensions and include the extra

Table 1.2: GPU microarchitectural parameters.

System Overview: No. of SMs: 15, Warp Size: 32 threads/warp; **Shader Core Config**: 1.4 GHz, GTO scheduler [22], 2 schedulers/SM; **Resources / SM**: No. of Warps: 48 Warps/SM, No. of Registers: 32,768; **Interconnect**: 1 crossbar/direction (15 SMs, 6 MCs), 700 MHz; **L1 Data Cache**: 16KB, 128B line, 4-way, LRU; **Shared Memory**: 48KB, 32 banks; **L2 Unified Cache**: 768KB, 128B line, 16-way, LRU; **Memory**: 6 GDDR5 Memory Controllers, 924 MHz, FR-FCFS [23]; **Bandwidth**: 177.4 GB/sec.

microarchitectural modifications necessary for the integration of neural accelerators within GPUs. The overhead of ISA extensions that communicate with the accelerator are modeled. For baseline simulations that do not include any approximation or acceleration, we use the unmodified GPGPU-Sim. We use one of the GPGPU-Sim's default configurations that closely models the Nvidia GTX 480 chipset with Fermi architecture. Table 1.2 summarizes the microarchitectural parameters of the chipset. We run the applications to completion. We use NVCC 4.2 with -O3 to enable aggressive compiler optimizations. Moreover, we optimize the number of thread blocks and number of threads-per-block of each kernel for the simulated hardware.

Energy modeling and overheads. To measure GPU energy, we use GPUWattch [19], which is integrated with GPGPU-Sim. To measure the accelerator energy, we also generate its event log during the cycle-accurate simulations . Our energy evaluations use a 40 nm process node and 1.4 GHz clock frequency. Neural acceleration requires the following changes to the SM and SIMD lanes and are modeled using McPAT [20] and CACTI 6.5 [21]. In each SM, we add a 2 KB weight FIFO. The extra input/output FIFOs are 256 bytes per SIMD lane. The sigmoid LUT which is added to each SIMD lane contains 2048 32-bit entries. Since GPUWattch also uses McPAT and CACTI, our added energy models, which use the same tools, provide a unified and consistent framework for energy measurement.

1.5.3 Experimental Results

Performance and energy benefits. Figure 1.4a shows the whole application speedup when all the invocations of the approximable region are accelerated with the neural accelerator. The remaining part (i.e., the non-approximable region) is executed normally. The results

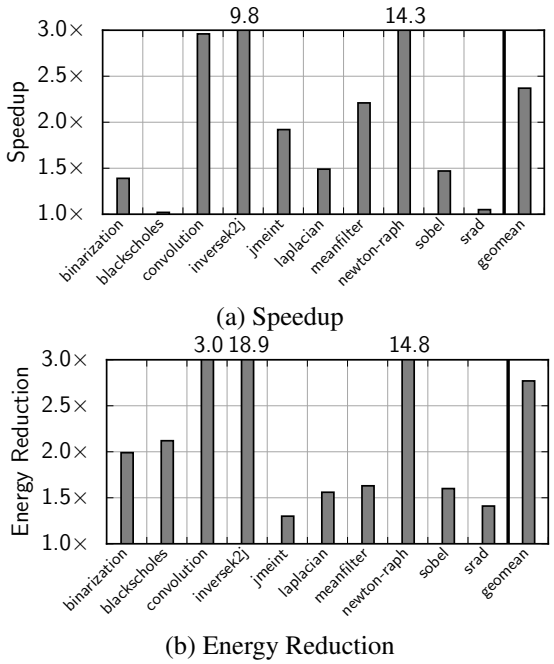


Figure 1.4: NGPU whole application speedup and energy reduction.

are normalized to the baseline where the entire application is executed on the GPU with no acceleration. The highest speedup is observed for newton-raph (14.3×) and inversek2j (9.8×), where the bulk of execution time is spent on approximable parts (see Figure 1.1). The lowest speedup is observed for blackscholes and srad (about 2% and 5%) which are bandwidth-hungry applications. While a considerable fraction of the execution time in blackscholes and srad is spent in the approximate region, the speedup of accelerating these two applications is modest. That is because these applications use most of the off-chip bandwidth, even when they run on GPU (without acceleration). Due to bandwidth limitation, neural acceleration cannot reduce the execution time. Next, we study the effect of increasing the off-chip bandwidth on these two applications and show that with reasonable improvement in bandwidth, even these benchmarks observe significant benefits. On average, the evaluated applications see a 2.4× speedup through neural acceleration.

Figure 1.4b shows the energy reduction for each benchmark as compared to the baseline where the whole benchmark is executed on GPU. Similar to the speedup, the highest energy saving is achieved for inversek2j (18.9×) and newton-raph (14.8×), where bulk of

the energy is consumed for the execution of approximable parts (see Figure 1.1). The lowest energy saving is obtained on jmeint (30%) since for this application, the fraction of energy consumed on approximable parts is relatively small (See Figure 1.1). On average, the evaluated applications see a 2.8× reduction in energy usage.

The quality loss when all the invocations of the approximable region get executed on neural accelerators (i.e., the highest quality loss) is shown in Table 1.1 (labeled Quality Loss). We study the effects of our quality control mechanism for trading off performance and energy savings for better quality later in this section.

Area overhead. To estimate the area overhead, we synthesize the sigmoid unit using Synopsys Design Compiler and NanGate 45 nm Open Cell library, targeting the same frequency as the SMs. We extract the area of the buffers and FIFOs from CACTI. Overall, the added hardware requires about 0.27 mm^2. We estimate the area of the SMs by inspecting the die photo of GTX 480 that implements the Fermi architecture. Each SM is about 22 mm^2 and the die area is 529 mm^2 with 15 SMs. The area overhead per SM is approximately 1.2% and the total area overhead is 0.77%. The low area overhead is because our architecture uses the same ALUs that are already available in each SIMD lane, shares the weight buffer across the lanes, and implements the sigmoid unit as a read-only lookup table, enabling the synthesis tool to optimize its area. This low area overhead confirms the scalability of our design.

1.6 Conclusion

Many of the emerging applications that can benefit from GPU acceleration are amenable to inexact computation. We exploited this opportunity by integrating an approximate form of acceleration, neural acceleration, within GPU architectures. Our neurally accelerated GPU architecture, provides significant performance and efficiency benefits while providing reasonably low hardware overhead (1.2% area overhead per SM). The quality control knob and mechanism also provided a way to navigate the tradeoff between the quality and the

benefits in efficiency and performance. Even with as low as 2.5% quality loss, our neurally accelerated GPU architecture (NGPU) provides average speedup of 1.9× and average energy savings of 2.1×. These benefits are more than 10× in several cases. These results suggest that *hardware* neural acceleration for GPU throughput processors can be a viable approach to significantly improve their performance and efficiency.

CHAPTER 2

SPECIALIZED COMPUTING STACK FOR DEEP LEARNING WITH FPGAS

This chapter is a result of work presented in MICRO 2016 [24].

2.1 Introduction

Deep Neural Networks (DNNs) are rapidly gaining traction in a wide range of applications such as vision, robotics, video analytics, speech recognition, natural language processing, targeted advertising, and web search [25, 26, 27, 28, 29, 30, 31, 32]. Although DNNs offer great prediction accuracy, they require a significant amount of computing power. With diminishing benefits from technology scaling [33, 34], the research community is increasingly turning to specialized accelerators for deep networks [35, 36, 37, 38] and other workloads [39, 40, 41, 42, 43, 44, 45]. Even though ASICs provide significant gains in performance and efficiency for DNNs [35, 36, 46, 47, 48, 49, 50], they may not cope with the ever-evolving DNN models. Furthermore, ASICs and customized cores come at the price of high non-recurring engineering costs over long design periods. Since FPGAs represent an intermediate point between the efficiency of ASICs and the programmability of general purpose processors, they are an attractive alternative for accelerating DNNs. Nonetheless, FPGAs still require extensive hardware design expertise and long design cycles. In fact, several research works [38, 37, 51, 52] have made extensive efforts to provide FPGA accelerators for specific DNN models, or parts of DNN computation, targeted for a particular FPGA platform.

Using FPGAs as an acceleration platform for DNNs is challenging as they offer a limited preset on-chip memory and often possess limited off-chip bandwidth, both of which are critical for high performance. This restriction is particularly limiting for FPGAs since ASIC designs can circumvent this issue by optimally allocating die area to on-chip mem-

ory for a single or set of target DNNs. The FPGA's memory and bandwidth limitations are further exacerbated for DNNs due to their high memory footprint, as well as high variability in the number of operations and model sizes for different DNN models (Section 2.8.1). A rigid accelerator architecture for DNNs may not fully utilize the FPGA's limited resources for every DNN model. It is thus essential co-optimize both the accelerator architecture and the corresponding execution schedule to overcome the FPGA's limited on-chip memory for each DNN model. This work seeks to provide such a solution by developing DNNWEAVER, a framework that generates synthesizable accelerators for a variety of FPGA platforms, while completely disengaging the programmers from hardware design. DNNWEAVER provides a comprehensive and automated solution to make FPGAs available to a broader community of DNN application developers who use a wide range of DNN models and often lack any hardware design expertise.

Table 2.1: Speedup and Performance-per-Watt comparison of DNNWEAVER generated accelerators. Each cell represents the benefits of the FPGA in row-heading relative to the platform in column-heading.

FPGA	ARM A15	Xeon E3	Tegra K1	GTX 650Ti	Tesla K40
Speedup Comparison					
Zynq	4.7×	0.59×	0.52×	0.15×	0.03×
Stratix V	22.39×	2.81×	2.43×	0.7×	0.15×
Arria 10	47.26×	5.94×	5.08×	1.48×	0.33×
Performance-per-Watt comparison					
Zynq	11.5×	16.6×	1.7×	3.2×	1.6×
Stratix V	5.5×	7.9×	0.8×	1.5×	0.8×
Arria 10	9.6×	13.9×	4.8×	2.7×	1.3×

These results show that DNNWEAVER generated accelerators outperform CPUs in performance and in two of three cases (Zynq and Arria 10) deliver higher Performance-per-Watt than GPUs. To achieve these benefits, the programmer only defines the topology and layers of the DNN (<300 lines of code) without dealing with hardware design or optimization. The relatively low programmer effort is particularly significant since the source code for our templates is over $10,000$ lines of code and is optimized hardware by experts over the course of one year. As Figure 2.1 illustrates, the DNNWEAVER generated accelerators for

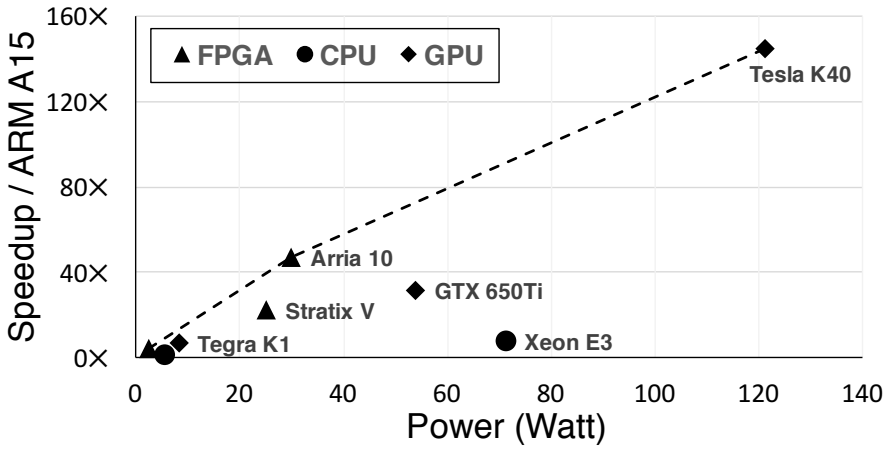


Figure 2.1: DNNWEAVER generated accelerators for Zynq and Arria 10 lie on the Pareto frontier (the dashed line). Tesla K40 represents the other Pareto optimal point. These results suggest that for high power setting GPUs are better programmable accelerators while DNNWEAVER makes FPGAs a compelling alternative when the power budget is limited.

Zynq and Arria 10 lie on the Pareto frontier. The Tesla K40 GPU represents the high-power high-performance Pareto optimal point. The results suggest that when power is limited, DNNWEAVER enables FPGAs to operate as a platform of choice for deep networks.

2.2 Contributions

This thesis chapter makes the following contributions to enable FPGA acceleration for a variety of DNNs:

(1) We develop a novel macro dataflow Instruction Set Architecture (ISA) for DNN accelerators. The ISA enables DNNWEAVER to expose a high-level programming interface. The programming interface is the same as Berkeley Caffe [53].

(2) Instead of just designing an accelerator for DNNs, we develop hand-optimized template designs that are scalable and highly customizable. The templates constitute a clustered hierarchical architecture that is contracted or expanded by DNNWEAVER to generate an accelerator that matches the needs of the DNN and the available resources on the FPGA.

(3) We provide a heuristic algorithm to co-optimize both the accelerator architecture and the corresponding execution schedule to minimize off-chip memory accesses and maximize performance. This algorithm strikes a balance between parallel operations and data

```
layer {                          layer {
 name: "conv1"                    name: "pool1"
 type: CONVOLUTION                type: POOLING
 bottom: "data"                   bottom: "conv1"
 top: "conv1"                     top: "pool1"
 convolution_param {              pooling_param {
   num_output: 20                   pool: MAX
   stride: 1                        stride: 2
   kernel_size: 5 } }               kernel_size: 2 } }
```

Figure 2.2: DnnWeaver programming interface.

reuse by slicing the computation and configuring the accelerator to best match the constraints of the FPGA.

Matching computation slice with the configuration of the accelerator is a unique challenge that needs to be addressed to create a framework that can generate highly efficient FPGA accelerators for DNNs. The aforementioned contributions enable DNNWEAVER to exploit the reconfigurability of the FPGAs while managing the large memory footprint of DNNs in the limited on-chip storage of FPGAs.

We use DNNWEAVER to generate accelerators for eight different deep networks targeted for three different FPGAs, Xilinx Zynq, Altera Stratix V, and Altera Arria 10. We rigorously compare the generated accelerators to both multicore CPUs (ARM A15 and Xeon E3) and many-core GPUs (Tegra K1, GTX 650Ti, and Tesla K40). The results are summarized in Table 2.1.

2.3 Overview of DNNWEAVER

This work seeks to alleviate the long design cycle necessary for using FPGAs to accelerate a wide variety of DNNs. We aim to create an automated framework that (1) completely dissociates programmers from the details of hardware design and optimization; (2) deals with the limited availability of on-chip resources (e.g., on-chip memory); and (3) provides a scalable and reusable FPGA acceleration framework, which delivers high performance and large efficiency gains for continuously evolving DNN models on different FPGA plat-

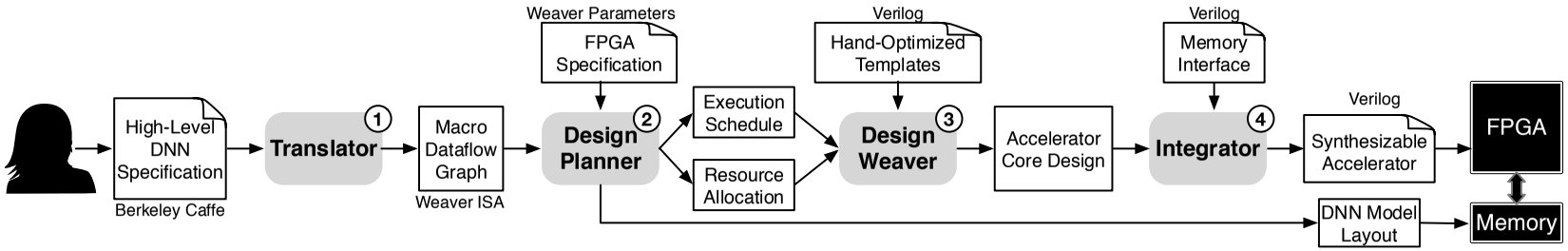


Figure 2.3: Overview of DNNWEAVER which takes as input high-level specification of a DNN and the target FPGA and generates the accelerator design as synthesizable Verilog along with the accelerator execution schedule and the layout of the DNN model in the memory.

forms. To achieve these conflicting objectives, we develop DNNWEAVER which combines hand-optimized scalable *template* designs with an *automated workflow* that customizes the templates to match the specifications of a given (DNN, FPGA) pair. The foremost task required by DNNWEAVER is that the programmer specifies the DNN models using a high-level programming interface as discussed below.

Programming interface. The input to DNNWEAVER is a high-level specification of the DNN in Berkeley Caffe format [53]. Caffe is a widely used open-source deep learning framework that takes the DNN specification as input and computes the given model on CPUs and GPUs. The code snippet in Figure 2.2 shows how two DNN layers, convolution and pooling, are described and connected in Caffe. Section 2.4 describes the functionality of DNN layers in detail.

As Figure 2.3 illustrates, DNNWEAVER automatically transforms the programmer-provided DNN model to an accelerator by generating FPGA synthesizable verilog code. DNNWEAVER comprises four software components; (1) the Translator, (2) the Design Planner, (3) the Design Weaver, and (4) the Integrator.

1. Translator. The Translator converts the DNN's specification to our macro dataflow instruction set architecture (ISA). Each instruction in the ISA represents a node in the macro dataflow graph of the DNN model. Note that the accelerator does not directly execute these instructions. DNNWEAVER compiler *statically* maps these instructions to control signals in the accelerator and creates an execution schedule. We choose this abstraction to provide a unified hardware-software interface and enable layer-specific optimizations in the accelerator microarchitecture without exposing them to the software. Hence, with this ISA, a variety of accelerator implementations which can be tuned to match the constraints of the target FPGA are possible. Furthermore, the ISA can be extended to support forthcoming layers or parameters. A one-time effort is required to develop the corresponding hardware templates. Section 2.5 describes this ISA in detail.

2. Design Planner. Design Planner accepts the instructions representing the macro dataflow

graph of the DNN and uses our Template Resource Optimization algorithm to optimize the hardware templates for the target FPGA platform. The Design Planner then partitions the computation of each layer to groups of operations that share and reuse data. We refer to each group's output as a slice. The slice is spilled to memory after computation and the accelerator proceeds to compute the next slice. Accelerating DNNs is particularly challenging due to their large memory footprint. By slicing its computations, DNNWEAVER manages this large footprint with the limited on-chip FPGA memory. Our Template Resource Optimization algorithm aims to strike a balance between parallel operations and data reuse by slicing computations and configuring the accelerator to best match the constraints of the FPGA (on-chip memory and external memory bandwidth). The Planner schedules slices of operations on the accelerator to generate a static execution schedule and the model layout in memory. Static scheduling simplifies the hardware and maximizes its efficiency and performance. Section 2.7 elaborates on the details of the Design Planner and our Template Resource Optimization algorithm.

3. Design Weaver. Design Weaver is the penultimate component of DNNWEAVER which takes as input the resource allocation and the execution schedule determined by the planner to generate the accelerator core. The Design Weaver uses a series of hand-optimized design templates and customizes them in accordance to the resource allocation and hardware organization provided by the planner. These templates provide a highly customizable, modular, and scalable implementation for the Design Weaver that automatically specializes the templates to accommodate a variety of DNN that are translated to our macro dataflow ISA. Furthermore, the Design Weaver converts the planner-provided execution schedule into state machines and microcodes, embedded within the hardware modules. Section 2.6 details the template designs and Section 2.7 discusses how the Design Weaver specializes the templates.

4. Integrator. The last component of DNNWEAVER is the Integrator, which adds the memory interface code to the accelerator code. As different FPGAs use different interfaces

to the external DRAM, the Integrator contains a library of DRAM interfaces and adds the appropriate code for each target FPGA. DNNWEAVER currently includes the DRAM interface for three series of FPGAs (Xilinx's Zynq, and Altera's Stratix V and Arria 10) from the two major vendors. After the integration, the final Verilog code is ready to be synthesized on the target FPGA to accelerate the specified DNN.

2.4 Background: Deep Neural Networks

The advent of deep learning, or more precisely, deep structured learning, can be traced back to Convolutional Neural Networks [54]. Convolutional Neural Networks are commonly used deep learning models, and hence are the focus of our work. As follows, a typical DNN consists of several back-to-back *layers* that represent increasingly abstract representations of the input.

Convolution layer. A convolution operation generates its output by sliding a window of parameters referred to as filters or kernels, over its inputs. A convolution layer is a set of these convolution operations that combine multiple input features and kernels to generate a single or multiple output feature maps. The initial layers of DNN are generally these convolution layers.

Pooling layer. A pooling layer down-samples its input to reduce its size. This layer sub-samples each window of the input feature maps to a single pooled output, which is usually the average, maximum, or minimum of the features in the window.

Inner product layer. This layer computes the inner product of an input vector and a weight matrix. Before computing this inner product, the previous layer output that might be multidimensional is arranged as a single dimensional vector.

Activation layer. An activation layer is a dimensionality preserving operation that applies an element-wise transfer function on its input feature map. Typical transfer functions are non-linear, (e.g., *sigmoid* , *tanh*), or piece-wise linear functions (e.g., *rectified linear*, *absolute value*).

Normalization layer. A normalization layer performs local inhibition by sliding a window over its input feature map. The normalization operation first produces the square-sum of the elements in the sliding window and then applies a non-linear function to the square-sum, which is multiplied with the input element being normalized to generate the output.

The programmer specifies the DNN using the layers described above. DNNWEAVER then converts this specification into a macro dataflow ISA that implements the operations of these layers as an abstraction for hardware acceleration. The details of this ISA are discussed in the next section.

2.5 Instruction Set Architecture Design

DNNWEAVER provides a macro dataflow ISA to (1) abstract away the details of accelerator design from the software; (2) enable layer-specific optimizations; (3) facilitate portability across different FPGA platforms; and (4) allow static execution scheduling at compile time. We chose a dataflow architecture to alleviate the von Neumann overhead of general-purpose architectures such as instruction fetch, decode, etc. The accelerator is not expected to execute these instructions. The compiler *statically* translates these instructions to microcodes and state machines. This dataflow architecture does not have explicit registers, which enables DNNWEAVER to impose significantly fewer restrictions on the accelerator architecture and allows portability across different FPGAs. Using an explicit dataflow architecture also allows DNNWEAVER to perform static optimizations at compile time and avoid data dependencies (e.g., register renaming) at runtime. Additionally, the coarse-grained nature of the ISA enables the microarchitecture to incorporate layer-specific optimizations without exposing them through the software stack.

Figure 2.4 shows the eight instructions of our ISA. These instructions are variable-sized and are designed to be able to express a large variety of deep neural networks. These instructions are further translated to state machines and microcodes at compile time. We use 64-bits to encode each word of the instruction. Since the ISA is a dataflow architecture,

Bits	63 – 60	59 – 56	55 – 32	31	30 – 24	23 – 17	16	15 – 10	9 – 4	3 – 0
input	Opcode = 0	Function	Destination Instruction ID	Fixed-Point/ Floating Point	Value Bitwidth	Fraction Bits/ Exponent Bits	1	# of Dimensions		
conv	Opcode = 1						Use the Next Word	Window Width	Window Height	Window Stride
pool	Opcode = 2									
norm	Opcode = 3									
ip	Opcode = 4							# of Neurons		
act	Opcode = 5						0	Reserved		
fanout	Opcode = 6		# Destinations							
output	Opcode = 7		0							

(a) IWORD1

Bits	63 – 48	47 - 32	31 – 16	15 – 0
input	Length of Dimension 0		Length of Dimension 1	
conv	Reserved	Window Width	Window Height	Window Stride
pool				
norm				
ip		# of Neurons		
act	Not Used			
fanout				
output				

(b) IWORD2 (Optional)

Figure 2.4: Instructions of the macro dataflow ISA.

each instruction is assigned a unique 24-bit static ID and none of the instructions include source operands. Instead, a part of instruction opcode encodes the unique static ID of the destination instruction that will receive the results. Below, we discuss each instruction type in our macro dataflow architecture.

Instruction `input`. The `input` instruction reads a DNN input (e.g., an entire image) from memory and sends it to another instruction for processing. As Figure 2.4(a) shows, bits `63--60` in the `IWORD1` contain the opcode, which in this case is `0`. Bits `59--56` are the `function` bits and are not used for this instruction. Bits `55--32` contain the unique ID of the destination instruction that will consume the inputs. Bit `31` indicates whether the generated values by this instruction are fixed-point or floating point. Bits `30--24` specify the total bit width of the generated values (e.g., 32). The bits `23--17` encode the number of fraction bits or the exponent, if the generated values are fixed point or floating point, respectively. Bit `16` indicates whether the next 64-bit word is part of the instruction. Bits

15--0 encode the number of dimensions in the DNN input (e.g., two for an input image.) The next instruction word, IWORD2, specifies the size of each dimension as presented in Figure 2.4(b). If the number of dimensions exceeds two, then more words are added to specify the size of dimensions. After specifying the size of all the dimensions, the next 64-bit value contains the address of the input in the memory.

Instruction `conv`. This instruction type performs the convolution operation by sliding a window over its inputs. The dimensions of the window and the sliding stride are encoded in bits 15-0, with six bits for the width and the height of the window, and four bits for the sliding stride. If these bits are not enough for specifying the window dimensions, bit 16 is set to 1 and IWORD2 is used to specify the structure of the window. The other fields in the IWORD1 are similar to the input instruction. After the IWORDs, an array of immediate values is used to specify the weights for the convolution operation.

Instruction `pool`. This instruction performs pooling on its inputs. Similar to conv, the structure of the pooling window and its stride is either specified in bits 15--0 or in IWORD2. The function field specifies the pooling type, such as max, min, or average.

Instruction `norm`. This instruction performs normalization and its window specification is similar to previous two instructions. The parameters are listed as immediate values after the IWORDs.

Instruction `ip`. This instruction corresponds to an inner product layer. Bits 15--0 specify the number of neurons in this layer, up to a maximum of 65536. For a larger number of neurons, IWORD2 is used. The immediate value after the IWORD is the address of the inner product weights in the memory.

Instruction `act`. This instruction corresponds to an activation layer and takes only one IWORD. The function field encodes the type of the activation function (e.g., sigmoid).

Instruction `fanout`. Since our ISA is dataflow and each instruction only encodes one destination, we provide a fanout instruction. This instruction is single IWORD and the bits 55--32 encode the number of destinations. The following immediate values after the

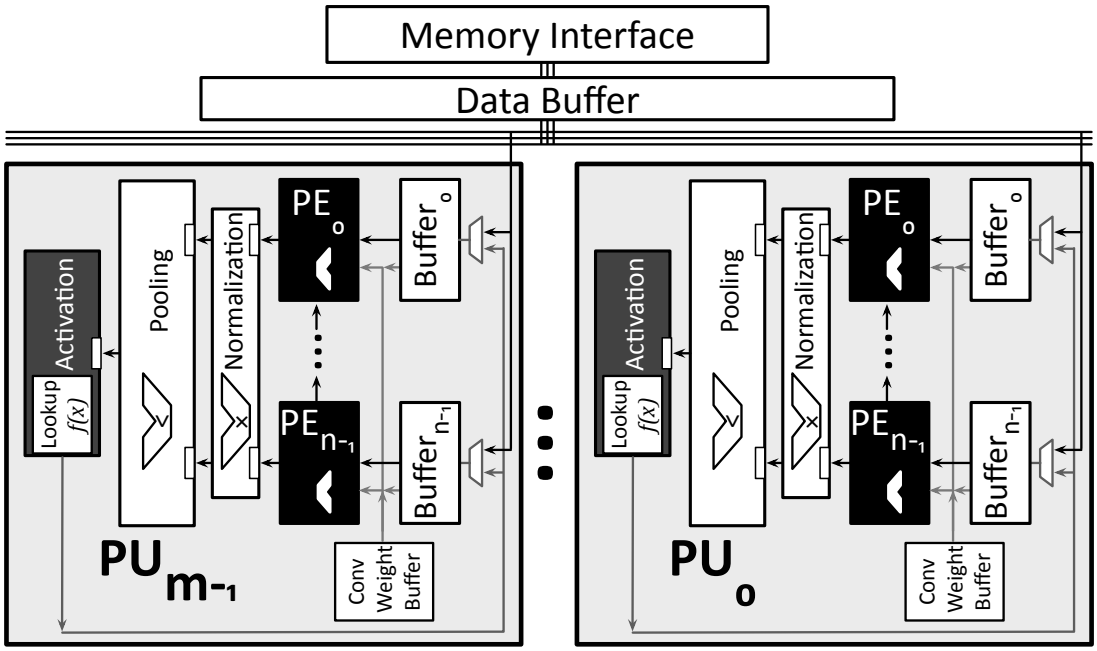


Figure 2.5: Overview of a clustered hierarchical template design. The template accelerator is divided into Processing Units (PUs) that are comprised of multiple smaller Processing Engines (PEs).

`IWORD` encode the ID of the destination instructions.

Instruction `output`. The output instruction does not have a destination instruction. It writes the outputs of the DNN to the memory address specified in the immediate values.

As discussed above, the instructions are translated to state machine and microcodes at compile time. Translation from Caffe to this ISA is straightforward since the instructions match the DNN layers. The Design Planner uses this ISA to customize the pre-designed templates and generate a static schedule for the operations within this macro dataflow, that is best suited for a given (DNN, target FPGA) pair. The next section discusses the hand-optimized template.

2.6 Template Accelerator Architecture

The template designs are highly *customizable* and *scalable*. The scalable architecture enables the Design Planner to shrink or expand the accelerator based on the requirements of the DNN and the resource constrains of the target FPGA. Templates are also designed to be *general*. That is, the templates include exchangeable components that realize differ-

ent layers of DNNs. If a DNN does not require a certain layer (e.g., normalization), the corresponding component is excluded to free resources for other layers.

2.6.1 Overall Organization

Figure 2.5 illustrates the template architecture that provides these necessary characteristics. As depicted, the template architecture is clustered with two levels of hierarchy; a collection of self-contained Processing Units (PUs) that comprise a set of smaller Processing Engines (PEs). The PEs and the buffers in the template PU architecture provide compute capabilities for convolution and inner product layers. The customizable normalization, pooling, and activation modules provide support for the other possible layers in DNNs. This clustered architecture provides scalability via modularity and by making the data traffic local to PUs and utilizing a unified bussing fabric across them. These features allow the Design Weaver to generate a concrete accelerator with any number of PUs and PEs-per-PU. Furthermore, the Design Weaver tunes the parameters of the hardware modules; all of which are extensively parameterized.

Specializing the design for a target FPGA. Each FPGA offers a certain number of hard blocks including DSP slices (ALUs) and Block RAMs (on-chip SRAM units, called BRAMs). Using these hard blocks is essential for exploiting the compute capabilities of the FPGA and achieving reasonably high frequency. Thus, the template architecture in Figure 2.5 maps the PU Buffers to the BRAMs and the ALUs to the DSP slices. The availability of these resources determines the maximum possible number of PEs and PUs for a given FPGA. However, as described in Section 2.7, the Design Planner determines the composition of the PU based on the size of the feature maps produced by the convolution/pooling/normalization/inner product layers to maximize resource utilization and the overall computation throughput. The next resource is the available off-chip bandwidth which determines the parameters of the `Data Buffer` that is connected to the memory interface as shown in Figure 2.5. The Design Planner performs static data marshaling and determines the layout

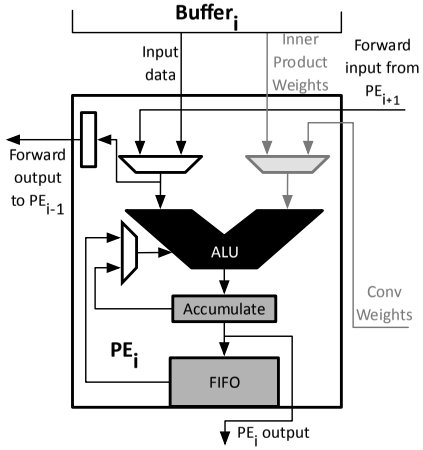


Figure 2.6: Processing Engine PE$_i$.

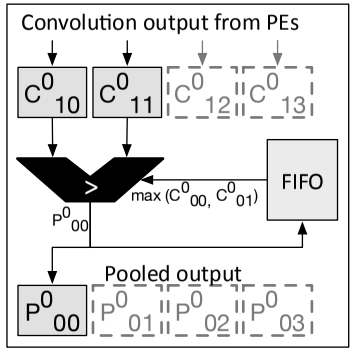


Figure 2.7: Pooling operations to compute P^0_{00}.

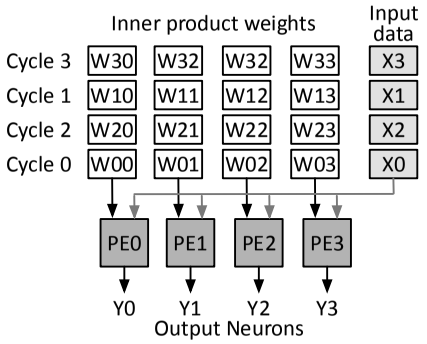


Figure 2.8: Execution of the Inner Product layer using MACC operations in PEs.

of the DNN weights and parameters to streamline transfer of parameters from the memory in contiguous chunks; maximizing the bandwidth utilization. The Design Planner also generates a static schedule for the `Data Buffer` to fetch data from the external memory and feed the PUs through the inter-PU bus. Static scheduling avoids contention on the bus and alleviates the need for PUs to perform complex handshaking. This approach, in turn, improves the scalability and efficiency of the template architecture.

Processing engines. PEs are the basic compute units that perform convolution, inner product, and parts of normalization. As shown in Figure 2.6, each PE contains a hard ALU that supports Multiply, and Multiply-Add operations. Neighboring PEs have a unidirectional link that forwards input data from a PE with higher index (PE_{i+1}) to the adjacent PE with lower index (PE_i). This forwarding link is used to reuse data across the adjacent PEs to minimize data transfer from memory. As depicted in Figure 2.5, each PE in a PU is associated with a dedicated buffer that feeds inner product weights and input data to the PE. Inner product weights typically require larger storage. These weights are streamed in the dedicated buffers that are mapped to the hard BRAM blocks.

Below, we use a running example (Figure 2.9) to demonstrate the operations and scheduling of different DNN layers.

2.6.2 Accelerating Layers of DNN

The first layer of the DNN in Figure 2.9 contains convolution kernels with a window size of 3×3 that produce two 8×8 outputs. Each convolution output is sub-sampled with a 2×2 max-pooling operation. Outputs from the pooling layer are arranged in a 1×32 vector, as the input to the inner product layer. The inner product layer has eight output neurons and requires a weight matrix of size 32×8. The example accelerator contains one PU with four PEs. For the running example, we assume that the model parameters are contained within the FPGA's on-chip storage. In Section 2.7, we relax this assumption and account for external memory accesses required when the data cannot fit on-chip.

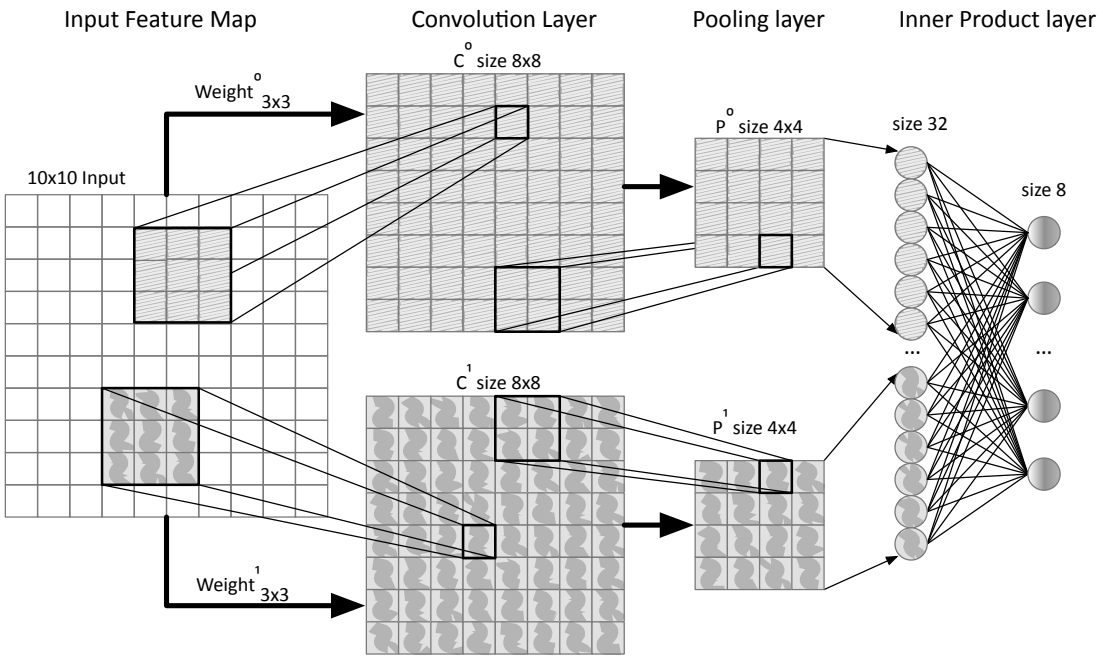


Figure 2.9: DNN example. Input elements are indexed as $X_{i,j}$.

Convolution Layer

As shown in Figure 2.9, the first layer convolves the input using two set of weights ($Weight^0$ and $Weight^1$) and produces two output feature maps (C^0 and C^1). The convolution operation can be expressed as a vector dot product between input elements and corresponding weights. The following operations are performed to generate the $\{0,0\}^{th}$ element of output feature map C^0.

$$C^0_{00} = Input_{00} \,.\, Weight^0$$

$$Input_{00} = [X_{00}, X_{01}, X_{02}, X_{10}, X_{11}, X_{12}, X_{20}, X_{21}, X_{22}]$$

$$Weight^0 = [W^0_0, W^0_1, W^0_2, W^0_3, W^0_4, W^0_5, W^0_6, W^0_7, W^0_8]$$

Dedicated buffer for convolution weights. To produce each output element in C^0, we require $Weight^0$. We minimize the overhead of accessing weights from the memory by using a convolution weight buffer in the PU that stores all the required weights and is shared across the PEs of the PU.

Parallelism across output elements. Convolution operations within an output feature map have no data dependencies on each other, and can be executed in parallel (e.g. C^0_{01}, C^0_{00},

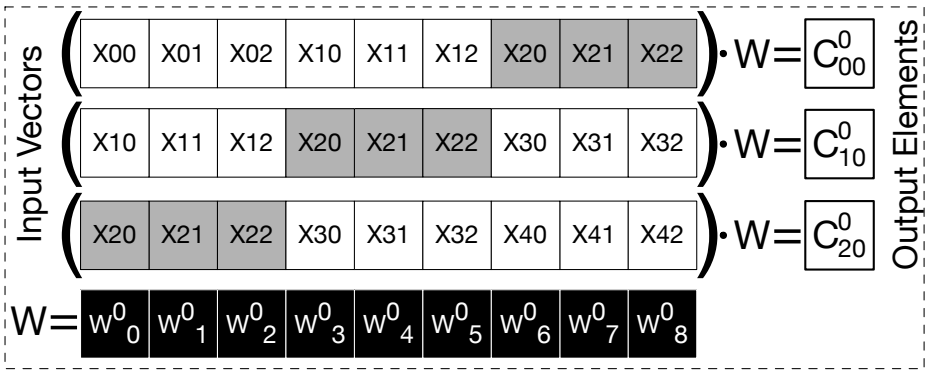


Figure 2.10: Convolution operations. X_{i0}, X_{i1}, and X_{i2} are input elements in the i^{th} row of the input feature map.

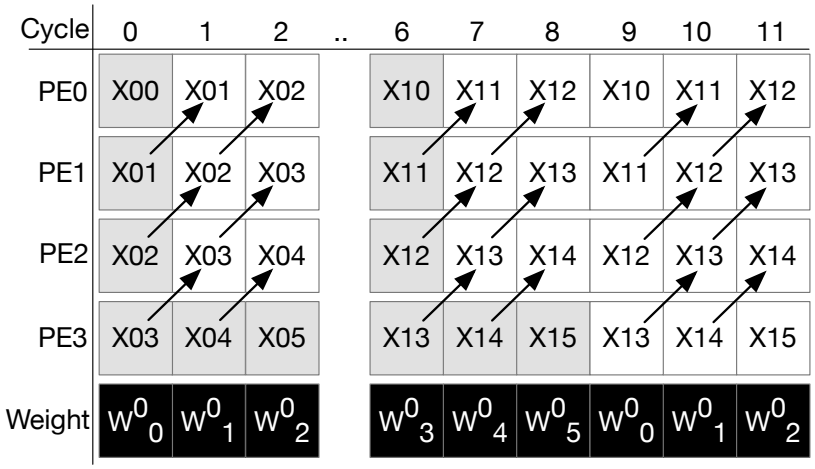


Figure 2.11: Convolution operation execution pattern.

etc.). These parallel calculations are performed by the PEs within a PU.

Saving partial results to minimize data communication. As shown in Figure 2.10, the convolution operations that generate the three outputs C^0_{00}, C^0_{10}, and C^0_{20} require accesses to the same inputs three times. These input elements $\{X_{20}, X_{21}, X_{22}\}$ are highlighted in gray in Figure 2.10. To reduce these redundant accesses, the PEs read the input row by row and generate partial results. The PEs then store the partial results in a local FIFO as depicted in Figure 2.6. When the PEs read the next set of the input elements, they also dequeue the partial results from the previous row and calculate the next set of partial results.

Data forwarding. Figure 2.11 shows another optimization that reduces remote data transfer through re-use. Convolution windows that produce adjacent outputs share input elements. Therefore, PEs computing adjacent output elements use partially shared data. We

add a dedicated unidirectional link between adjacent PEs to forward these shared input elements. The arrows in Figure 2.11 show this data forwarding to re-use data for convolution. The unique data read accesses for each PE are highlighted in gray.

Reusing data across convolution kernels. Using the sequence of operations in Figure 2.11, the four PEs compute four adjacent output elements using the first kernel ($Weight^0$).

Pooling Layer

The example uses a window of size 2×2 and a stride of two for max pooling. The input feature map for the pooling layer is divided into 16 non-overlapping windows of size 2×2, each corresponding to an output element. As Figure 2.7 illustrates, to compute pooling output element P^0_{00}, the unit require $C^0_{00}, C^0_{01}, C^0_{10}, C^0_{11}$. Since the convolution layer produces adjacent elements in a row, the unit first compute max(C^0_{00}, C^0_{01}) for the first row and push it to the FIFO. When the second row is available, the FIFO is popped and we compute max(max(C^0_{00}, C^0_{01}), C^0_{10}, C^0_{11}).

Hiding latency. To hide execution latency, the pooling module overlaps its operations with the convolution operations in PEs. Since the kernel size for convolution in the previous layer is 3×3, the four PEs in the PU generate four output elements every nine cycles. As shown in Figure 2.7, the convolved output elements are sent to the shared pooling module with a single $3:1$ comparator.

Inner Product layer

Inner product layer can be expressed as a vector-matrix multiplication. In the running example, the pooling layer produces two outputs, P^0 and P^1 of size 4×4 each or 32 elements in total. P^0 and P^1 are flattened and concatenated to generate a 1×32 input vector that is multiplied with the weight matrix of size 32×8 to generate the output vector of size 1×8, shown in Figure 2.9.

Parallelism across output elements. In inner product layer, each output neuron Y_j is

generated as $Y_j = \sum_i X_i * W_{ij}$, where X_is are the inputs to the layer and W_{ij}s are the weights. To exploit the parallelism between output computations, each output neuron Y_j is assigned and computed on a single PE using a series of multiply-accumulate operations. With four PEs, the PU simultaneously calculates outputs Y_j in groups of four starting from $\{Y_0, Y_1, Y_2, Y_3\}$ as shown in Figure 2.8.

Normalization and Activation

As Figure 2.5 depicts, a part of normalization (sum of squares) uses the convolution hardware and the other part (scaling) is implemented as a separate unit. The activation transfer functions are implemented using lookup tables in each PU.

The Design Planner exploits the FPGA's reconfigurability by customizing the described template architecture for the target FPGA and DNN model as discussed in the following section.

2.7 Design Planner

The template architecture described in the previous section serves as a scalable template for the accelerator's microarchitecture. DNNWEAVER takes advantage of the FPGA's reconfigurability using Template Resource Optimization, a heuristic search algorithm, that co-optimizes both the accelerator architecture and the corresponding execution schedule to minimize off-chip accesses and maximize performance. The two key factors affecting performance are: (1) the allocation of compute and memory resources to the various components in the template architecture, which determines the degrees of parallelism in the accelerator; and (2) the schedule of operations on the accelerator, which determines the required external memory accesses. This algorithm aims to strike a balance between parallel operations and data reuse and configuring the accelerator to best match the constraints of the FPGA (on-chip memory and memory interface bandwidth). As the memory requirement of DNNs is typically much higher than the on-chip storage available on FPGAs, we

divide the output feature map of each layer into slices. A slice is a portion of the output feature map that is spilled to memory after computation. Template Resource Optimization maximizes performance by varying (1) PEs-per-PU and (2) slice dimensions. Below, we discuss the two variables in further detail.

Variable (1) Number of PEs-per-PU. The template architecture exposes two levels of parallelism: (1) parallelism between PEs in a PU that generate adjacent output elements, and (2) parallelism between PUs generating independent output feature maps. Due to a fixed number of resources on the FPGA, increasing the number of PEs-per-PU would decrease the total number of PUs and vice versa. Our Template Resource Optimization algorithm aims to find a PEs-per-PU configuration that strikes a balance between the two degrees of parallelism and provides the best performance for the (DNN, FPGA) pair.

Variable (2) Output slice. The next variable is the slice of the output feature map that is computed within each epoch of the PU execution. This slice is the fraction of the output feature map that fits in the on-chip storage of the PU. The amount of this storage depends on the number of PEs[1] in each PU. The slicing dictates the number of external memory accesses and determines the degree of reuse in the computation. Convolution-like layers operate on overlapping input elements. Reusing these overlapping elements can only happen within a slice but not across slices. Therefore, the slice determines the degree of data reuse. The algorithm first tries to fit a row of the output in the on-chip memory of the PU. The stride is based on the number of PEs-per-PU to match the outputs with the PEs. If extra storage is still available, it tries to store more output rows. With this approach, the Design Planner picks a slice that maximizes data reuse and minimizes external memory accesses for each PU. By doing so, all the output slice computations can be done with local information in the PU. Another aspect of this approach is that the next layer can start operating on the slice before spilling it to memory. This optimization further reduces the off-chip memory accesses.

[1]Note that each PE is assigned a BRAM.

Template Resource Optimization search algorithm. Algorithm 1 illustrates our heuristic search which solves the following optimization objective:

$$arg\ \min\ executionCycles\ =\ \sum_{l} f(nPEperPU\ ,\ sliceSize_l)$$

$$\text{subject to}\quad 1\ \leq nPEperPU\ \leq \text{FPGA.max\_PEs}$$

$$sliceSize_l\ \leq min(BRAM \times nPEperPU\ ,\ output_{size})$$

Here, $nPEperPU$ is the number of PEs-per-PU and $sliceSize_l$ is the dimensions of the slice in layer l, FPGA.max_PEs is the maximum PEs that can the accommodated on the FPGA, and $output_{size}$ is size of a single channel in the output feature map.

Inputs : D: DNN Macro Dataflow Graph
F: FPGA Constraints
Output : $nPEperPU$: number of PEs per PU
$sliceSize_l$: Slice dimension in each layer
arg min: eec: $execution\_cycles$
Function findSliceSize*(pe, F)*
  Initialize $width_r = pe$ Initialize $height_c = 1$
  while *($width_r \times height_c \leq$ min (F.BRAM $\times$ pe, $output_{size}$)* **do**
    $width_r = width_r$ + *pe*
  end
  while *($width_r \times height_c \leq$ min (F.BRAM $\times$ pe, $output_{size}$)* **do**
    $height_c = height_c$ + 1
  end
  return $width_r$, $height_c$
end
Initialize $eec = \infty$; Initialize l = $D.numLayers$
Initialize $nPEperPU = 1$; Initialize $\forall\ sliceSize_l$ in $l = \phi$
for *pe in range (1,*$F.max_{PE}$*)* **do**
  Initialize $\forall\ ss_l$ in $l = \phi$
  for $\forall\ l$ *in* l **do**
    ss_l = findSliceSize(*pe*, F)
  end
  $cycles = g(D, F, pe, ss_l)$
  if *(cycles ¡ eec)* **then**
    $eec = cycles$
    $nPEperPU = pe$
    $\forall\ l$ in L $sliceSize_l = ss_l$
  end
end

Algorithm 1: Template Resource Optimization search.

The algorithm takes in as input the DNN macro dataflow graph (D) and the constraints of the FPGA platform (F). The FPGA constraints (F) provide the maximum number

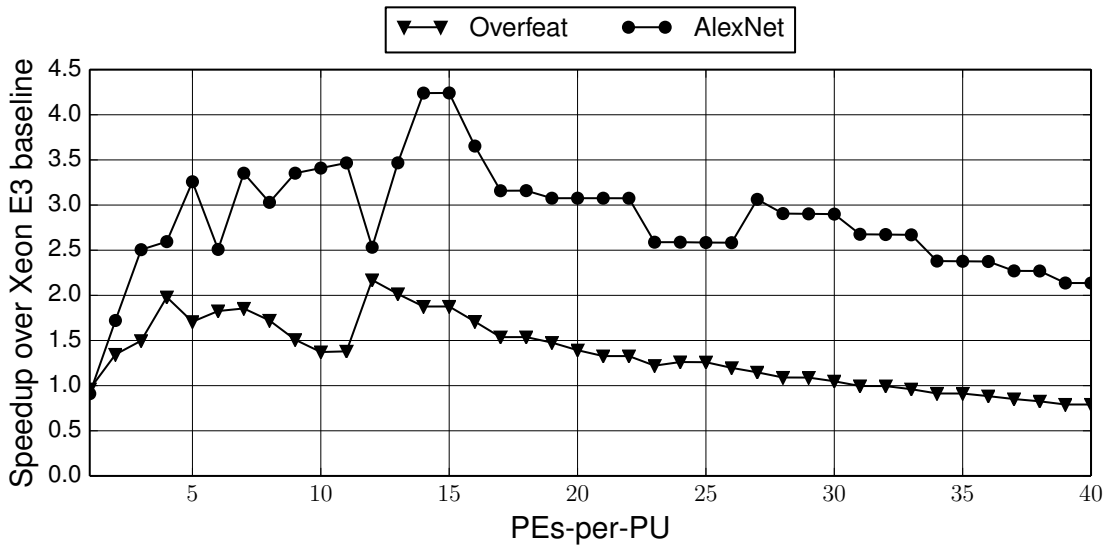


Figure 2.12: Design space exploration for optimizing resource allocation.

of PEs and the capacity of the BRAM in each PE. The algorithm finally outputs the $nPEperPU$ and the $sliceSize_l$ by taking the following steps:

(1) Initialize. Initialize the number $nPEperPU$ and $sliceSize_l$ (for each layer). Initialize the estimated[2] execution cycles (eec) to ∞. The eec is an estimation of the number of cycles to execute a particular DNN with an organization that complies with $nPEperPU$ and $sliceSize$.

(2) Increment $nPEperPU$**.** Vary the $nPEperPU$ iteratively starting from 1 to the maximum number of PEs that can be synthesized on the FPGA platform.

(3) Calculate $sliceSize_l$**.** For the current choice of the $nPEperPU$, calculate the dimensions of the slice that fits in the PU. This calculation is done for each layer of the DNN.

(4) Estimate execution cycles. Estimate the execution cycles given the current choices of $nPEperPU$ and $sliceSize_l$.

(5) Reiterate or terminate. If the $cycles$ is less than eec, record the choices. Terminate if $nPEperPU$ has reached the maximum value, otherwise reiterate from step (2).

Figure 2.12 illustrates the result of search for Altera's Arria10 FPGA for AlexNet [55] and

[2]We have built a mathematical model g to estimate the execution cycles.

Overfeat [56]. We use Xeon E3 as the baseline to better visualize the trends. Performance for a layer in a DNN is highest when the output feature map size is a multiple of PEs-per-PU. Thus, the PEs-per-PU configuration that achieves best performance varies for the layers in a DNN, resulting in multiple peaks as shown in Figure 2.12. The peak performance occurs at 14 PEs-per-PU for AlexNet and 12 PEs-per-PU for Overfeat. The $sliceSize_l$s are different for each point of the graph.

2.8 Evaluation

To evaluate the effectiveness of DNNWEAVER, we use three off-the-shelf FPGA platforms, Xilinx Zynq ZC702, Altera Stratix V GS D5, and Altera Arria 10 GX-AX115. Table 2.2 summarizes their specifications. Henceforth, we refer to DNNWEAVER generated accelerator for Zynq, Stratix V, and Arria 10 as DW-Zynq, DW-Stratix, and DW-Arria, respectively.

2.8.1 Methodology

Benchmark DNNs and Their Input Datasets

Table 2.3 shows our benchmark DNN models, their input datasets, size of model parameters, and the number of multiply-accumulate operations required. The selected DNNs are used for various applications ranging from handwritten digit recognition, object recognition, to speech-to-text decoders. Among these, CIFAR-10 Full targets object detection in the CIFAR-10 thumbnail dataset [57]. LeNet targets the MNIST handwritten digit recognition dataset [58]. The DjiNN ASR network is a DNN speech-to-text decoder obtained from the DjiNN and Tonic benchmark suite [30]. NiN [59], AlexNet [55], Overfeat [56], VGG-CNN-S [60], and VGG-16 [60] target the ILSVRC ImageNet dataset [61].

CPU and GPU Execution

Table 2.4 lists the five evaluated CPU and GPU platforms.

Table 2.2: FPGA Platform Details.

	Xilinx Zynq ZC702	Altera Stratix V SGSD5	Altera Arria 10 GX115
FPGA Capacity	53K LUTs	172K LUTs	427K LUTS
	106K Flip-Flops	690K Flip-Flops	1708K Flip-Flops
Peak Frequency	250 MHz	800 MHz	800 MHz
BRAM	630 KB	5035 KB	6782 KB
MACC Count	220	1590	1518
Evaluation Kit Price	$895	$6,995	$4495
Technology	TSMC 28nm	TSMC 28nm	TSMC 20nm

Table 2.3: Benchmark DNNs and their input datasets. The model size provides the size in Mega Bytes of the weights required for the network.

Network	Data set	Domain	Model Size (MegaBytes)	Multiply-Accumulate Operations
CIFAR-10 Full	CIFAR-10	Object Recognition	0.17 MB	12,390,400
LeNet	MNIST	Handwritten Digit Recognition	0.82 MB	2,293,000
NiN	ImageNet	Object detection and classification	14.50 MB	1,105,996,320
Djinn ASR	Kaldi	Speech-to-text decoder	48.40 MB	25,366,528
AlexNet	ImageNet	Object detection and classification	116.26 MB	736,332,416
VGG-CNN-S	ImageNet	Object detection and classification	196.26 MB	2,666,222,720
Overfeat	ImageNet	Object detection and classification	278.30 MB	2,797,535,776
VGG-16	ImageNet	Object detection and classification	323.87 MB	16,361,995,456

Table 2.4: Evaluated CPUs and GPUs.

Platform	Cores	Clock freq (GHz)	Memory (GB)	TDP (W)	Technology (nm)	Cost
ARM Cortex A15	4+1	2.300	2 (shared)	5	28	$191
Intel Xeon E3-1246 v3	4	3.600	16	84	22	$290
Tegra K1 GPU	192	0.852	2 (shared)	5	28	$191
NVIDIA GTX650Ti	768	0.928	1	110	28	$150
Tesla K40	2880	0.875	12	235	28	$2,599

Runtime measurements. We compare the execution time of DNNWEAVER generated accelerators to the execution time on CPUs and GPUs using Berkeley Caffe. The CPU and GPU baselines are compiled with GCC 4.8 and NVCC 6.5, respectively. We obtain the baseline timings by using the timing feature of Caffe. For Arria 10 and Stratix V, we synthesize the accelerator using `Qaurtus II v14.1` tool and use a cycle estimator to measure performance for the synthesized accelerator. Across all other platforms, we run each DNN `100` times and use the average.

Multi-threaded vectorized CPU execution. We use `OpenBLAS` for the BLAS backend required by Caffe to produce CPU-specific optimized binaries. Hence, we used Haswell-specific optimization for the Xeon E3 CPU, and the A15 optimization for the ARM A15 processor (Jetson TK1). The Haswell version of OpenBLAS uses AVX2 and Fused Multiply-Add (FMA) instructions whereas the A15 version uses the NEON SIMD engine. When evaluating the Xeon E3 CPU we used 4 threads, as we empirically found that this provided the best performance – enabling SMT affected the performance negatively. For the ARM A15 CPU we used 4 threads as well since it does not have SMT support.

Optimized GPU execution with cuDNN. For fastest GPU execution, Caffe can be configured to use the NVIDIA `cuDNN` library. We use the latest `cuDNNv5` for GTX 650Ti and Tesla K40, and `cuDNNv2` for Tegra K1, separately compiled for each GPU with architecture-specific compiler optimization. Tegra K1 does not support newer versions of `cuDNN`.

FPGA platforms details. In the Zynq board, the interface between DRAM and programmable logic is a standard AXI bus. In the Arria 10 and Stratix V boards, we used Altera's Avalon interface IP for interfacing the DRAM with the programmable logic. We implement a custom controller on the programmable logic to interface with the main memory. We synthesize the hardware with 64-bit `Vivado v2015.1` for the Zynq board and `Qaurtus II v14.1` for the Stratix V and Arria 10 boards. We use the synthesis tools to generate the area utilization numbers presented in Table 2.5. The frequency of operation of the accelerator on the Zynq board is `150 MHz` and the frequency of operation on the Stratix

V and Arria 10 FPGAs is 200 MHz.

Power measurements using vendor libraries. We employ a variety of strategies in order to gather power measurements for most tested platforms. We use the NVIDIA Management Library (NVML) to obtain the average power of Tesla K40. Given that GTX 650Ti does not support the NVML library, and since the GTX 650Ti and Tesla K40 share the same microarchitecture, we make a conservative estimation of the GTX 650Ti power by scaling the Tesla K40 measurements using the two chips' TDPs. For each DNN, we calculated the ratio of measured power in Tesla K40 over its TDP. We multiply this ratio with the GTX 650Ti TDP and use the 95% of this number.

We utilize the Intel Running Average Power Limit (RAPL) energy consumption counters available in the Linux kernel for power measurements on the Xeon E3.

Power measurements in hardware. The ARM A15 CPU and the Tegra K1 GPU are a part of the Jetson TK1 development board. Jetson TK1 does not provide software means to gather power readings. Therefore, we use the Keysight E3649A Programmable DC Power Supply to get the power consumption of the Jetson TK1 board. To do so, we subtract the idle average power 3.12W from the power reading we obtain during benchmark execution. For Arria 10 and Stratix V, we use the TDP as a measure of power consumed during execution. Finally, we use a GPIO to USB adapter to read the power directly from the power controllers in the Zynq board.

2.8.2 Comparison to High Level Synthesis

As an alternative to DNNWEAVER, HLS can also generate hardware implementations for DNNs, where the programmer uses HLS's C-like syntax to express layers of the DNN model. Although HLS provides a high level abstraction to programmers, optimizing the hardware implementation for a DNN model on a target FPGA requires expertise in both hardware design and the specific programming tool used for HLS. In our experiments, two Masters students with moderate amount of experience in hardware design spent one month

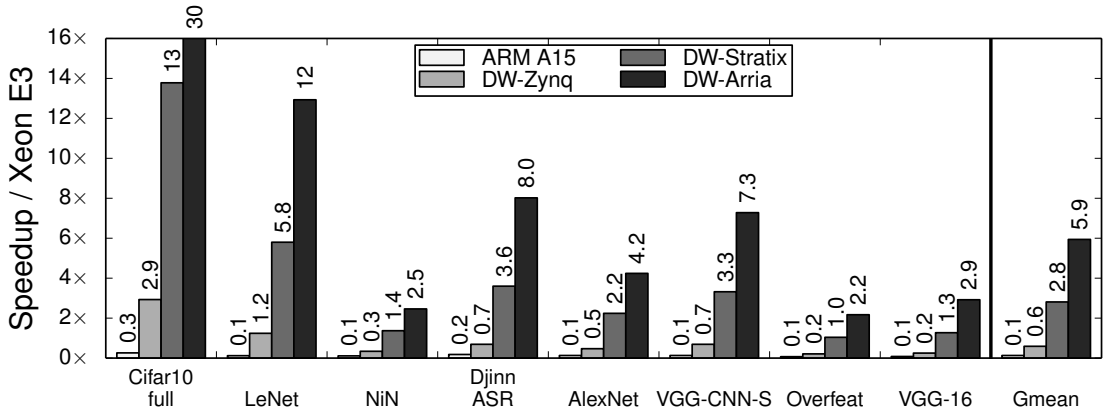


Figure 2.13: Speedup of DNNWEAVER generated accelerators in comparison to CPUs (baseline=Xeon E3)

to optimize a Vivado HLS implementation of the LeNet Benchmark for the Xilinx Zynq ZC702 FPGA. The resulting implementation ran at 100 MHz and provided a slow-down of 19.7× compared to DNNWEAVER generated accelerator for the same FPGA platform. The benefits of the template approach is more evident when considering a recent work [37] that uses commercial HLS tool and yet spends significant effort to implement the convolution layers of just one DNN, AlexNet. Moreover, another recent work [62] shows that using dataflow templates as an intermediate compilation target for high-level synthesis of C/C++ programs delivers 9× higher performance than the state-of-the-art HLS tools. Recall that for none of the FPGA acceleration, DNNWEAVER requires anything beyond just expressing the DNN in Caffe format.

2.8.3 Experimental Results

Performance Comparison with CPUs

Speedup compared to Xeon E3. Figure 2.13 illustrates the performance benefits when the DW-Zynq, DW-Stratix, and DW-Arria are used to compute the models under evaluation. The performance of Xeon E3 for the eight DNN models using Caffe's framework is used as a baseline for comparison. The average speedup for DW-Zynq, DW-Stratix, and DW-Arria is 0.59×, 2.81×, and 5.94×, respectively; thus, DW-Arria provides 10× more speedup than DW-Zynq. Among the evaluated models, Cifar-10 Full sees the highest speedup of

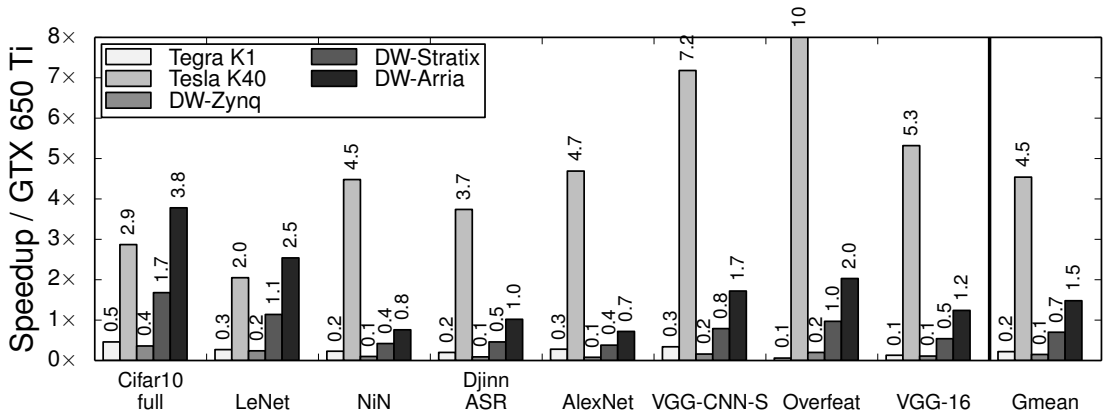


Figure 2.14: Speedup of DNNWEAVER generated accelerators in comparison to GPUs (baseline=GTX 650Ti)

(2.9×, 13.8×, and 30.9×) while Overfeat shows the lowest speedup of (0.2×, 1.0×, and 2.2×) over DW-Zynq, DW-Stratix, and DW-Arria, respectively. The significant gap in performance benefits comes from the disparity in the model topology, some layers are more favorable to the DNNWEAVER generated accelerators than the others. We will further discuss the difference in per-layer computation efficiency later in Section 2.8.3.

Speedup compared to ARM A15. Figure 2.13 also shows the performance comparison with a low-end processor, ARM A15. ARM A15 exhibits an 8× slowdown with respect to Xeon E3. Compared to the low-end ARM A15 processor, DW-Zynq, DW-Stratix, and DW-Arria provide 4.7×, 22.39×, and 47.26× speedup respectively. As expected, the low power ARM A15, which is not intended for high performance computing is significantly outperformed by the Xeon E3 server class processor.

These results demonstrate the performance benefits provided by DNNWEAVER generated accelerators over both low-end and high-end CPUs, as well as their scalability over various FPGAs.

Performance Comparison with GPUs

Speedup compared to GTX 650Ti. We compare our accelerators with GPU platforms including GTX 650Ti, Tegra K1, and Tesla K40 in Figure 2.14. The baseline is GTX 650Ti, a middle-tier GPU. DW-Zynq and DW-Stratix provide 6.60× and 1.4× slowdown compared

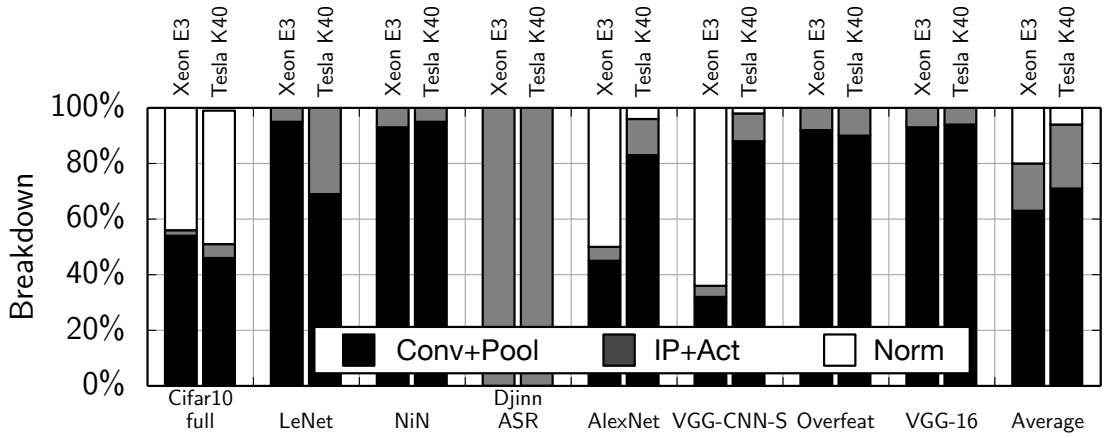


Figure 2.15: Runtime breakdown across the DNN layers for Xeon E3 and Tesla K40. (Conv: Convolution, Pool: Pooling, IP: Inner Product, Act: Activation, Norm: Normalization).

to GTX 650Ti, while DW-Arria shows a 1.48× speedup. Tesla K40 provides a speedup of 4.5× over the baseline. For the three FPGAs, (DW-Zynq, DW-Stratix, and DW-Arria), maximum speedup of (0.4×, 1.7×, and 3.8×) is observed from Cifar-10 Full, whereas AlexNet shows the minimum speedup of (0.1×, 0.4×, and 0.7×).

Speedup compared to Tegra K1 and Tesla K40. In Figure 2.14, we also show a comparison with a low-end GPU, Tegra K1, and a high-end GPU, Tesla K40. The low-end Tegra K1 offers a 0.2× average speedup over the baseline. In contrast, the high-end GPU Tesla K40 presents a 4.5× speedup. In comparison with Tesla K40, (DW-Zynq, DW-Stratix, and DW-Arria) show an average speedup of (0.03×, 0.15×, and 0.33×), respectively.

Per-Layer Performance Benefits

To understand the per-layer efficiency of the DNNWEAVER generated accelerators, we examine the runtime breakdown across the model layers and the speedup for individual layers.

Runtime breakdown. Figure 2.15 shows the runtime breakdown of the models computed from the Caffe framework using two baselines platforms, Xeon E3 and Tesla K40. For convenience, we combine (1) Convolution and Pooling layers (Conv+Pool), (2) Inner Product and Activation layers (IP+Act), and (3) Normalization layer (Norm). On average, Conv+Pool occupies 68% and 74% of the computation runtime, while the IP+Act occupies 16% and

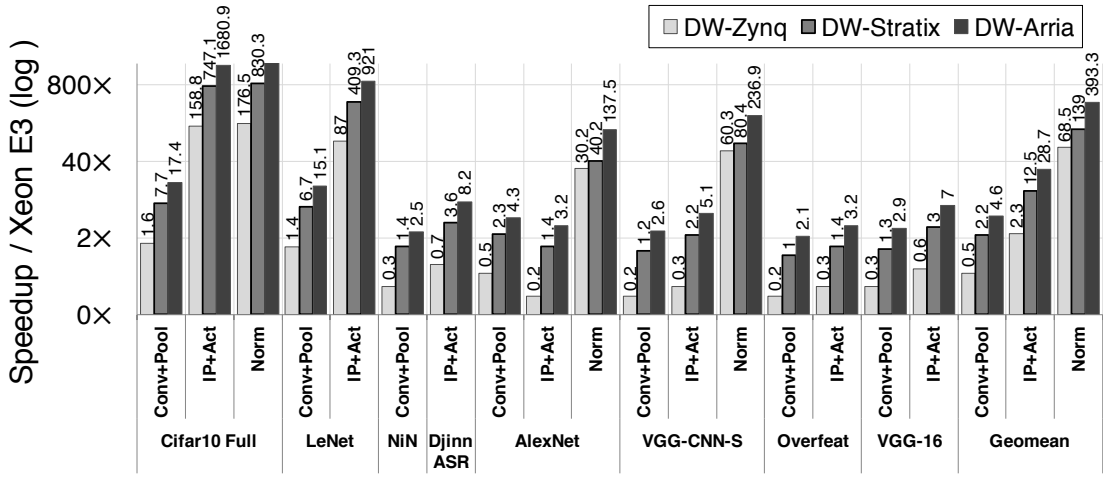


Figure 2.16: Speedup for each DNN layer with the baseline of Xeon E3.

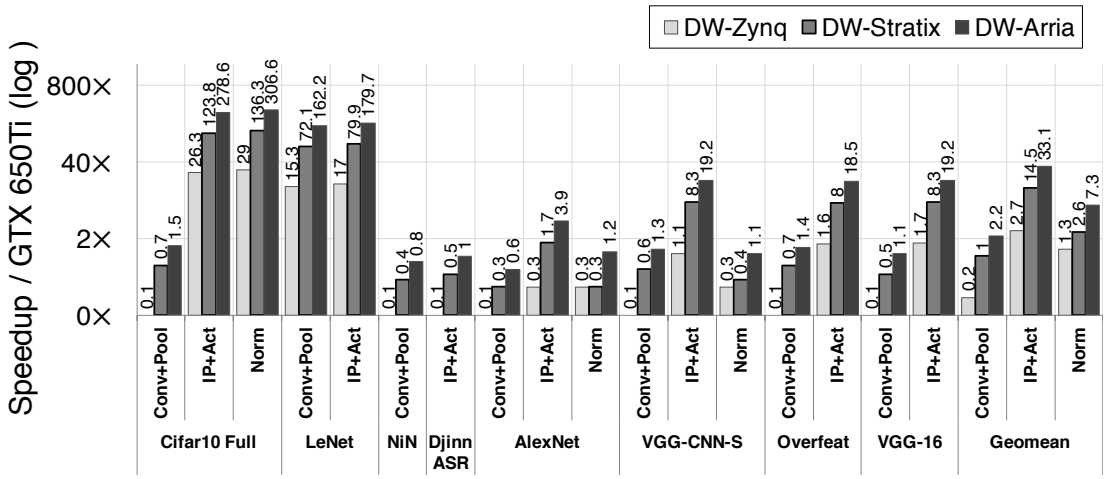


Figure 2.17: Speedup for each DNN layer with the baseline of GTX 650Ti.

21% when run on Xeon E3 and Tesla K40, respectively. The larger proportion of execution time is spent on Conv+Pool than the other layers as the convolution layer has significantly higher number of operations than the inner product layer.

The composition of runtime varies between the network models depending on the network topology and layer sizes. Cifar10 Full, AlexNet, and VGG-CNN-S are the only networks that include a normalization layer, which is executed for 16% and 5% of the runtime on Xeon E3 and Tesla K40, respectively. With the exception of Djinn ASR, which consists of just IP+Act, most of the time in the rest of the benchmarks is spent on Conv+Pool.

Per-layer speedup. Figure 2.16 shows the per-layer speedup of the DNNWEAVER generated accelerators compared to CPUs with the baseline software execution as Xeon E3. As

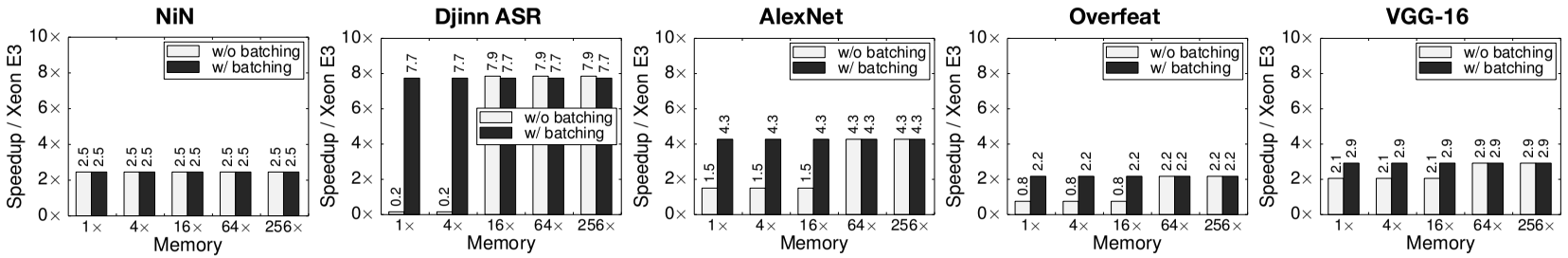


Figure 2.18: Speedup over Xeon E3 when varying the available on-chip storage. We use a validated cycle-accurate simulator to generate these results.

shown in Figure 2.16, for the set of (DW-Zynq, DW-Stratix, DW-Arria), the average speedup for Conv+Pool, IP+Act, and Norm in comparison with Xeon E3 is (0.5×, 2.2×, and 4.6×), (2.3×, 12.5×, and 28.7×), and (68.5×, 139×, and 393×), respectively. Norm shows high speedup, particularly over CPUs, as the non-linear operations within normalization are implemented efficiently in FPGAs using lookup tables. Norm is a significant portion of the runtime for Cifar10 Full and VGG-CNN-S, leading to a high speedup for these models. Overfeat's runtime is dominated by Conv+Pool, which presents the lowest speedup.

Similarly, Figure 2.17 shows the per-layer speedup with the baseline of GTX 650Ti. For Norm and IP+Act, DW-Stratix outperforms the baseline, while it closely follows the baseline performance for Conv+Pool. DW-Arria also follows the same trend and the speedup for IP+Act is higher than that for Norm.

Sensitivity to on-chip storage

Figure 2.18 illustrates the impact of limited on-chip storage over performance on the Arria 10 board. We use a cycle accurate simulator, which we validate against hardware measurements, to generate the figure. The figure compares the performance of DNNWEAVER generated accelerator to Xeon E3 CPU baseline when varying the size of on-chip BRAM resource from 1× to 256× the available. The impact of memory size is most prominent for inner product layers, since the inner product weights are significantly large, and seldom fit in on-chip memory. Storing inner product weights off-chip reduces the accelerator performance due to the overhead of external memory accesses. As illustrated in Figure 2.18, the amount of on-chip storage required to store the inner product weights and overcome this memory wall is more than 16× the available storage.

We reduce this overhead by sharing inner product weights over a batch of inputs. Figure 2.18 compares the performance of the generated accelerator with and without batching. The speedup from batching is most prominent in Djinn ASR, where the model consists of back-to-back inner product and activation layers. On the other hand, the performance of

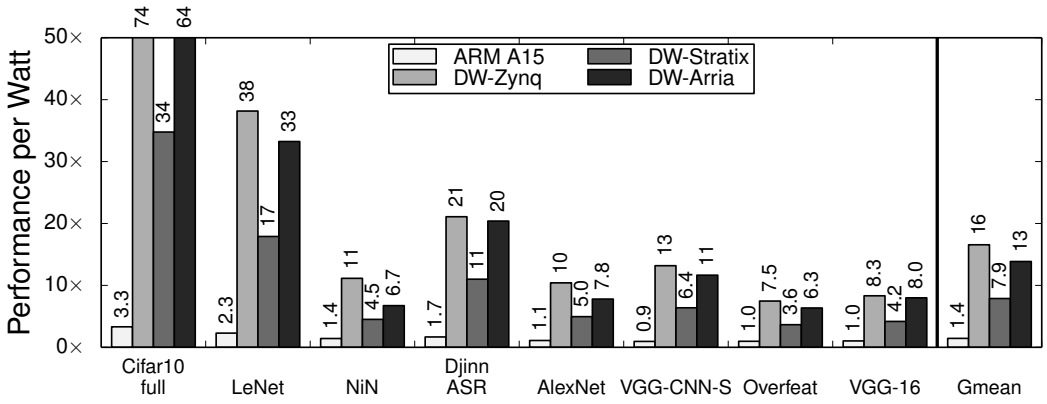


Figure 2.19: CPU Performance-per-Watt Comparison (Baseline=Xeon E3)

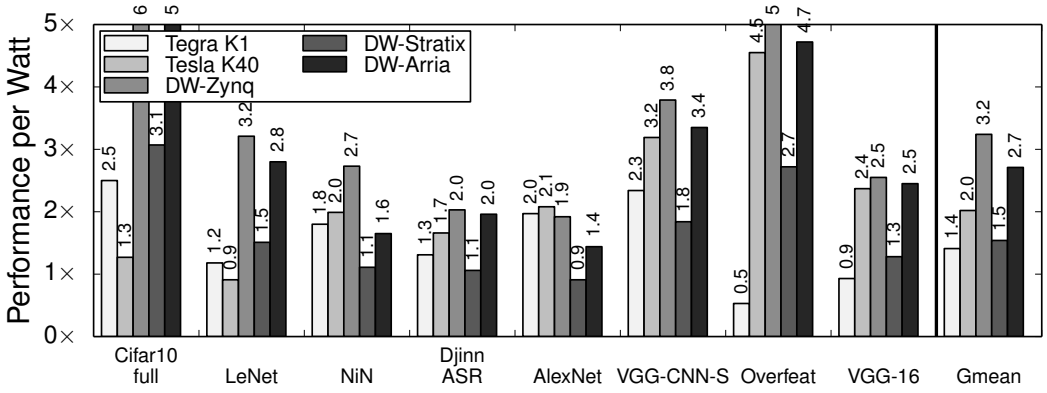


Figure 2.20: GPU Performance-per-Watt Comparison (Baseline=GTX 650Ti)

NiN is unaffected by batching as it does not contain an inner product layer. The benchmarks AlexNet, Overfeat, and VGG-16 observe a $2.2\times$ increase in performance through batching. Benchmarks VGG-CNN-S, LeNet and Cifar10 Full observe a similar trend.

Performance-per-Watt Comparison with CPUs

As shown in the speedup results, the performance benefits from diverse CPU, GPU, and FPGA platforms vary substantially. In fact, these hardware platforms occupy different design points in the underlying performance vs. energy efficiency tradeoff. To understand the performance benefits with the fixed energy efficiency, we measure the power consumption and evaluate the performance-per-Watt for each hardware platform.

Comparison with Xeon E3. Figure 2.19 compares the performance-per-Watt for ARM A15, DW-Zynq, and DW-Arria with the baseline of Xeon E3. On average, DW-Zynq shows $16.6\times$,

DW-Stratix shows 7.9×, and DW-Arria shows 13.9× higher performance-per-Watt than the baseline. Note that although DW-Stratix provides about 10× higher speedup, the increased power consumption by DW-Stratix (2W vs. 25W) leads to the lower performance-per-Watt than DW-Zynq. However, DW-Arria provides higher performance that DW-Stratix, without a commensurate increase in power consumption, leading to higher performance-per-Watt. This trend is observed for all the evaluated DNN models.

Comparison with ARM A15. Low-end processors such as ARM A15 are commonly used in mobile devices and are known to have high energy-efficiency. We also compare the ARM A15 processor with our accelerators and Xeon E3. The ARM A15 processor shows 1.4× higher performance-per-Watt compared to Xeon E3. When compared with ARM A15, DW-Zynq, DW-Stratix, and DW-Arria show 11.5×, 5.5×, and 9.6× higher performance-per-Watt, which demonstrates the energy efficiency of the DNNWEAVER generated accelerators.

Performance-per-Watt Comparison with GPUs

Comparison with GTX 650Ti. Figure 2.20 shows the performance-per-Watt in comparison of Tegra K1, Tesla K40, DW-Zynq, and DW-Arria with the baseline, GTX 650Ti. The pair of (DW-Zynq, DW-Stratix, and DW-Arria) provides (3.2×, 1.5×, and 2.7×) higher performance-per-Watt than the baseline. Although DW-Arria outperforms DW-Zynq with the speedup of 10× shown in Figure 2.14, DW-Zynq offers a 1.2× higher performance-per-Watt compared to DW-Arria.

Comparison with Tegra K1 and Tesla K40. Figure 2.20 also compares the performance-per-Watt of Tegra K1 and Tesla K40 with the baseline. On average, Tegra K1 and Tesla K40 have 1.4× and 2.0× higher performance-per-Watt than GTX 650Ti.

Table 2.5: Resource utilization on the three FPGA platforms for each benchmark DNN.

Benchmark DNN	Xilinx Zynq ZC702				Altera Stratix V SGSD5				Altera Arria 10 GX115			
	LUTs (Total: 53200)	BRAM (Bytes) (Total: 630KB)	Flip-Flops (Total: 106400)	DSP Slices (Total : 220)	LUTs (Total:	BRAM (Bytes) (Total: 5035KB)	Flip-Flops (Total: 690000)	DSP Slices (Total: 1590)	LUTs (Total:	BRAM (Bytes) (Total: 6782KB)	Flip-Flops (Total: 1708800)	DSP Slices (Total: 1518)
	Utilization	Utilization	Utilization	Utilization	Utilization	Utilization	Utilization	Utilization	Utilization	Utilization	Utilization	Utilization
Cifar-10 Full	61.44%	98.57%	30.77%	61.82%	85.29%	95.33%	46.90%	37.74%	84.99%	84.92%	45.85%	94.86%
Djinn ASR	42.43%	100.00%	18.25%	63.64%	53.98%	85.80%	28.12%	36.23%	68.96%	94.36%	39.27%	98.81%
LeNet	47.57%	100.00%	21.90%	61.82%	66.53%	80.44%	32.83%	33.96%	84.99%	84.92%	45.85%	94.86%
VGG_CNN_S	62.22%	97.14%	29.73%	61.82%	88.07%	78.50%	50.81%	37.04%	84.64%	89.64%	47.59%	88.54%
VGG_16	65.92%	100.00%	31.23%	63.64%	87.65%	78.20%	50.68%	37.42%	84.64%	89.64%	47.59%	88.54%
AlexNet	64.56%	100.00%	30.78%	63.64%	86.70%	77.16%	50.04%	37.23%	82.19%	86.69%	46.24%	88.54%
NiN	68.59%	100.00%	34.62%	63.64%	86.70%	77.16%	50.04%	37.23%	84.64%	89.64%	47.59%	88.54%
Overfeat	61.52%	94.29%	29.28%	60.00%	84.68%	75.07%	48.79%	36.98%	85.57%	86.25%	48.19%	88.93%

Table 2.6: Total number of PUs and the number of PEs per PU built on the three FPGA platforms for each benchmark DNN.

	Xilinx Zynq ZC702		Altera Stratix V SGSD5		Altera Arria 10 GX115	
Benchmark	**# of PUs**	**# of PEs**	**# of PUs**	**# of PEs**	**# of PUs**	**# of PEs**
Cifar10 full	8	17	8	60	8	135
Djinn ASR	7	20	23	24	23	54
MNIST LeNet	8	17	8	60	8	135
VGG_CNN_S	17	8	17	31	19	64
VGG_16	10	14	15	35	19	64
AlexNet	14	10	14	37	14	84
NiN	7	20	14	37	19	64
Overfeat	12	11	12	42	13	90

Area and FPGA Utilization

Table 2.6 shows the framework determined number of PUs and the number of PEs-per-PU for DW-Zynq, DW-Stratix, and DW-Arria. The resource utilization in DW-Stratix is limited by the LUTs available on chip, and the resource utilization in DW-Zynq is bounded by the number of BRAM blocks available on-chip. Table 2.5 shows the resource utilization to generate the DNNWEAVER accelerators for each DNN model.

2.9 Related Work

There have been several proposed and realized hardware designs that accelerate machine learning algorithms and DNNs. However, this work differs from other efforts in that DNNWEAVER is not an accelerator, but an accelerator generator. Our work produces an optimized design for a given (DNN, FPGA) pair. Furthermore, DNNWEAVER provides a novel ISA to unify DNN accelerators across different FPGA platforms. In this section we discuss the most related work in the area of FPGA implementations and ASIC accelerators for DNNs.

FPGA implementations for machine learning. Tabla [45] provides an FPGA accelerator generator for the training phase of statistical machine learning algorithms. However, DNNWEAVER focuses on inference with DNNs. In addition, Tabla uses stochastic gradient descent as the abstraction between hardware and software, and has no notion

of ISA or Deep Neural Networks. Tabla provides its own mathematical language, while DNNWEAVER uses Berkeley Caffe for model specification.

The work by Chen, et al. [37] focuses on using an analytical design scheme based on the roofline model to find the fastest design for a particular DNN for FPGA acceleration. However, their design does not support some DNN layers such as pooling and normalization. The work by Farabet, et al. [38, 52] develops an FPGA accelerator for a specific DNN. Gokhale, et al. [51] propose a mobile co-processor for DNNs and evaluate it on a Zynq board. Chakradhar, et al. [63] present a VLIW co-processor for DNNs and emulate it on a Virtex 5 FPGA. They propose a special switch that allows to dynamically group the convolution engines in different ways. The design has a low-level VLIW ISA but the paper does not include any details about its design. Unlike DNNWEAVER, they do not generate Verilog code for FPGA accelerators. The works by Qiu et al. and Suda et al. [64, 65] present implementations of accelerators for particular DNN models. Neither of these works support generation of accelerators for arbitrary DNN topologies.

DNNWEAVER makes FPGAs accessible to the machine learning community by automatically generating an optimized accelerator from high level DNN specifications. On the other hand, previous works come short of providing at least one of the following features: optimized accelerator generation, ISA support, a workflow starting from high level abstractions.

ASIC accelerators for DNNs. Recent research efforts present low-power deep learning ASICs. For example, (Da)Diannao [35, 36] provide DNN accelerators with a low-level fine-grained ISA, yet they do not define an ISA to unify DNN accelerators. In contrast, DNNWEAVER uses a ISA for deep neural networks (DNN) representing high-level operations (layers) that provides the flexibility necessary to optimize the accelerator microarchitecture for the FPGA platform and DNN model. Sim, et al. [48] showcase a DNN ASIC for IoT devices. However, the article doesn't make a reference to classification layer support. Qadeer, et al. [49] propose Convolution Engine which reduces the number of operations

required in convolution layers. Conti, et al. [50] develop convolution cores designed to integrate with a shared-memory cluster of RISC processors. PuDianNao [43] is an ASIC accelerator for machine learning algorithms but lacks deep convolutional networks support.

All of these previous efforts require ASIC design, not FPGA realization, which is the focus of our work.

Concurrent submissions. Hardware implementation for DNNs is a thriving and active area of research. The following efforts have been published concurrently to our work. Wang, et al. [66] use a library of fixed-function blocks to accelerate DNNs on Xilinx Z7020 and Z7045 FPGAs. Unlike the PEs in DNNWEAVER, the architecture in their work lack explicit data sharing. Liu, et al. [67] propose an ISA for neural networks optimized for high code density over vector and matrix operations. Chen, et al. [46, 47] develop an ASIC design with a 2D spatial array of PEs for Convolutional Neural Networks. Song, et al. [68] propose an ASIC implementation with adaptive data-level parallelism for DNN accelerators. EIE [69], Minerva [70], and Cnvlutin [71] propose ASIC accelerators that use operation pruning and quantization in DNNs for power and performance benefits.

2.10 Conclusion

Deep Neural Networks are gaining increasing applicability and are amongst the most important workloads that can significantly benefit from acceleration. However, DNNs are in a state of flux and new disruptive advances require hardware solutions that can adapt to these changes. DNNWEAVER is an initial step in providing such solutions that support a wide variety of DNN models and can be further extended for more advanced models. While GPUs serve as an attractive platform for DNNs, our results shows that FPGAs can be a Pareto optimal choice when power is constraining. Nonetheless, reducing the programmer involvement in hardware design is imperative to the adoption of FPGAs in this domain. To this end, DNNWEAVER converts high-level specification of DNNs into highly efficient accelerators that operate within a limited power budget and on-chip memory of the FPGA.

The conversion is made possible by a novel dataflow ISA and a heuristic search algorithm that generates high performance accelerator by customizing the hand-optimized template designs for a given (DNN, FPGA) pair. DNNWEAVER takes an effective step in making FPGAs available to a broader community of DNN developers who often do not possess hardware design expertise. Community engagement and contribution are vital for providing a general platform for DNN acceleration. To facilitate such engagement, DNNWEAVER has been made publicly available at http://dnnweaver.org.

CHAPTER 3

ACCELERATING MACHINE LEARNING TRAINING AT CLOUD SCALE

3.1 Introduction

Prevalence of interconnected compute platforms has transformed the IT industry, which is now rapidly moving towards scale-out solutions that extract insights from data. Following this trend, systems that enable distributed computing on general-purpose platforms are gaining eminence (e.g., Spark [72] and Hadoop [73]). In a concurrent yet disjoint effort, due to the diminishing benefits from general-purpose processing, the community is developing mostly single-node accelerators for a variety of applications, including machine learning [35, 36, 44, 43, 70, 69, 46, 71, 24, 45]. However, there is a gap between scale-out systems and accelerators due to the lack of solutions that enable distributed acceleration at scale. Moreover, it is not enough to just design and integrate accelerators independent from algorithms and programming interfaces. We need a holistic approach that reworks the fundamental hardware-software abstractions and enables a broad community of programmers to seamlessly utilize accelerators at scale for a specific domain of applications. Reusing the traditional stack for scale-out acceleration is inadequate as the entire computing stack is designed and optimized merely for CPUs, which were the sole processing platform up until recently. To that end, this paper sets out to design a full and specialized computing stack, dubbed CoSMIC[1], for scale-out acceleration of learning.

CoSMIC offers the entire stack of layers to execute a wide range of learning algorithms on accelerator-augmented scale-out systems. These layers comprise a domain-specific language, a compiler, a specialized runtime system, and a multi-threaded template architecture for the accelerator. The template architecture can be automatically tailored for deployment on FPGAs or realization as custom Programmable ASICs (P-ASICs). FPGAs offer flex-

[1] **CoSMIC**: **Co**mputing **S**tack for **M**L acceleration **I**n the **C**loud

ibility as well as efficiency and are becoming readily available in different markets [74, 42, 75, 76], now even in Amazon Elastic Compute Cloud (EC2) [76]. Not only have FPGAs become a lower-cost alternative to ASICs, but also serve as prototypes for custom chip design. However, designing efficient accelerators is onerous even when targeting a single-node FPGA and requires extensive expertise in both hardware design and application domain. This challenge is exacerbated in the scale-out setting due to the added complexity of task distribution and communication. Additionally, P-ASICs impose high non-recurring engineering costs over long design periods and usually need unintuitive or narrow programming interfaces. Furthermore, as technology is scaled, modern FPGAs and ASICs can harbor an ample amount of resources, whose effective utilization necessitates rethinking accelerator design paradigms. Therefore, to realize scale-out acceleration, we address the following triad of challenges when devising the CoSMIC full stack: (1) efficiently exploiting large number of on-chip resources, (2) enabling distributed acceleration using accelerator-augmented nodes, and (3) relieving programmers of distributed system coordination and the onus of hardware design. Furthermore, CoSMIC targets a wide class of learning algorithms and provides support for new learning models and algorithmic changes to the existing ones. To realize CoSMIC we were required to address the following research challenges.

(1) How to enable scale-out acceleration of many ML algorithms, yet disengage programmers from hardware design.

To tackle this challenge, CoSMIC leverages a combination of two theoretical insights: (1) a wide range of learning algorithms are stochastic optimization problems, solved using a variant of gradient descent [77, 78, 45]; (2) differentiation is a linear mathematical operator, and thus the gradient over a set of data points can be calculated as an aggregated value over the partial gradients computed in parallel for each point [79, 80, 81, 82, 83, 84, 85]. A variety of learning algorithms can be parallelized using these two insights. Examples include, but are not limited to, recommender systems, Kalman filters, linear and nonlin-

ear regression models, support vector machines, least square models, logistic regression, backpropagation, softmax functions, and conditional random fields. To implement these algorithms, one needs to have (1) the partial gradient calculation function, (2) the aggregation operator, and (3) the number of data points that are processed before each aggregation. The first layer of the CoSMIC stack exposes a high-level mathematical language to programmers to specify these three constructs, which capture the entirety of the learning algorithm. The next layer of the CoSMIC stack fully automates the scale-out acceleration. The CoSMIC compiler maps and schedules the operations on the distributed accelerators. The next layer, a specialized runtime system, assigns roles and tasks for the scale-out system components and orchestrates the distributed calculation of the partial gradients and their iterative aggregation. The final layer of the CoSMIC stack provides a novel multi-threaded template architecture for the accelerators. This layer can be automatically customized and tailored according to the high-level specification of the learning algorithm and the constraints of the system.

(2) How to design customizable accelerators that efficiently exploit the large capacity of advanced process technologies.

Advanced manufacturing processes have made integration of compute and storage resources on the chip. As a result, even modern FPGAs offer large capacities—e.g. Intel Arria 10 [86] instances comprise 1,518 DSP slices with 6.6 MBytes of storage and Xilinx UltraScale+ in Amazon EC2 [76] includes 6,840 DSP slices and 43 MBytes of storage. A single instance of learning algorithm may not effectively exploit resources since it is limited by the fine-grained parallelism in its Dataflow Graph (DFG). Therefore, CoSMIC offers a novel Multiple-Instruction Multiple-Data (MIMD) multi-threaded template architecture that divides the resources across multiple instances of the learning algorithm as independent threads. The last layer of CoSMIC customizes this template and generates the final accelerator by striking a balance between the number of threads running on the chip and the resources assigned to each thread. The code generation differs for FPGAs and P-ASICs.

For FPGAs, the generated core is tailored to one specific learning algorithm as the chip can be erased and reprogrammed for different applications. For P-ASICs, the generated accelerator is a programmable superset of the design that fits in the area and power budget of the chip. Any algorithm that can be expressed using the DSL can be compiled and accelerated on the generated P-ASIC. The generated code and template are in the form of Register-Transfer Level (RTL) Verilog code. The template architecture is designed, optimized, and implemented by experts once in Verilog, which ensures efficiency although CoSMIC generates the accelerators automatically. More specifically, the template is designed as a two-dimensional matrix of compute units to ensure data dependencies and within-thread communications do not curtail its scalability to rather large number of processing elements. We also designed a tree-like bus to connect the rows and allocated bidirectional communication across columns. Hence, the communication latency only grows by a logarithmic order with an increase in the number of compute units, improving on-chip scalability. Furthermore, CoSMIC's backend compiler minimizes data movement by mapping operations to where their operands are located. This hardware-software co-design that aims to maximize effective resource utilization ensures effective utilization of on-chip resources, especially when they are plentiful.

(3) How to devise the system software that is specialized for distributed multi-threaded acceleration of learning.

To be inline with the recent industry trends in integrating accelerators in datacenters [42, 75, 76], CoSMIC targets commodity distributed systems in which accelerators sit on the high-speed expansion slots (e.g., PCIe). For generality, we assume no special connectivity between the accelerators although such connectivity will most likely improve the benefits of CoSMIC. CoSMIC aims to best utilize the system-wide resource on both CPUs and accelerators. CoSMIC achieves this objective by offering a lean and specialized system software layer that exclusively supports learning algorithms that can be trained using parallel variants of stochastic gradient descent. This specialized layer allows the CoSMIC stack to

assign the partial gradient calculation onto the accelerators while the CPUs perform aggregation and networking. This task assignment alleviates the use of accelerator resources for TCP/IP communication, avoids data copies to accelerator boards for aggregation, and enables using commodity distributed systems with CoSMIC. Moreover, it maximizes system-wide resource utilization as well as portability to different accelerator boards. Within each node, the system software maintains an internal thread pool. These threads handle the communication with the remote peer nodes. Internally managing this thread pool avoids costly OS-level context switches. The system software layer also maintains another internal thread pool that asynchronously aggregates the partial gradients. In addition, this layer assigns roles to the nodes and orchestrates the exchange of partial gradients and their aggregation.

We evaluate the benefits of the CoSMIC stack using 10 different learning applications from various domains including medical diagnosis, computer vision, finance, audio processing, and recommender systems. We compare CoSMIC against Spark, a popular framework for scale-out computing using the optimized MLlib machine learning library [87]. On average, a 16-node CoSMIC with UltraScale+ VU9P FPGAs offers 18.8× speedup over a 16-node Spark system with Xeon E3 Skylake CPUs while the programmer only writes 22–55 lines of code. When scaling the nodes from 4 to 16, CoSMIC's performance improves by 2.7×, while Spark's performance scales only by 1.8×. We also compare the CoSMIC system with the distributed GPU (NVIDIA Tesla K40c) implementation. We report the benefits of CoSMIC for two P-ASIC implementations that match the compute resources and off-chip bandwidth of the FPGA and the GPU. On average, these P-ASICs offer 1.2× and 2.3× higher system-wide performance, while the GPU delivers 1.5× speedup over FPGA system. While using custom chips can improve computation time by 11.4×, the system-wide performance benefits are limited to 2.3×. Finally, with CoSMIC's novel multi-threaded accelerator architecture, the FPGA and the two P-ASIC systems respectively achieve 4.2×, 6.9×, and 8.2× higher Performance-per-Watt than the GPU system. These results confirm

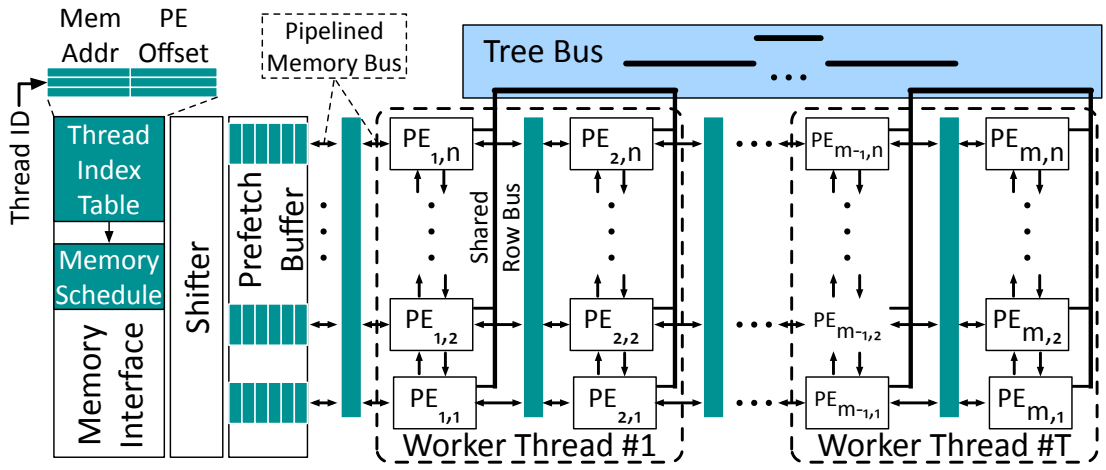


Figure 3.1: CoSMIC Multi-Threaded Template Architecture.

that CoSMIC is an effective and vital initial step to enable acceleration of learning at scale. To this end, this work not only contributes the full stack of CoSMIC, but also defines a new multithreaded accelerator architecture, a novel communication-aware scheduling and mapping algorithm, and a lean and specialized system software for thread management and system orchestration.

3.2 CoSMIC Template Architecture

A major challenge in acceleration is the generality across a wide range of algorithms and applications while supporting a variety of platforms (e.g., various FPGA chips). It is also crucial to offer a solution that can adapt to new algorithms and algorithmic changes. A fixed architecture cannot offer enough flexibility and is not deployable on different chips. Therefore, CoSMIC offers a template architecture to accelerate learning at scale. This template is predesigned, yet re-organizable, providing the capability to implement different gradient calculations and parallel variants of gradient descent aggregations and updates. The template offers reusability while delivering high performance, as it is hand-crafted by experts (e.g., our team). Our stack stretches and squeezes the template to best match the DFGs and the target chip. Hence, it is modular and scalable to maximally utilize the ample amount of resources in the server-grade FPGAs and P-ASICs.

The need for multi-threading. A single instance of a learning algorithm cannot effectively exploit as much resources, since it is limited by the level of parallelism in its DFG. The DFG of the partial gradient update dictates the number and type of operations, along with data-dependencies. However, data-dependencies in the DFG limit the number of operations that the accelerator can execute in parallel. To increase the parallelism available to the accelerator, we use the insight that partial gradient updates generated by worker threads in parallel gradient descent algorithms are independent. As such, the CoSMIC template architecture executes multiple worker threads in the FPGA accelerator; each thread, using a subset of the accelerator resources, executes the entire DFG over the thread's data sub-partition to generate an independent partial gradient update. This multi-threading limits the data-communication within a worker thread to a subset of the accelerator's DSP slices, reducing communication overhead.

3.2.1 Accelerator Organization

As depicted in Figure 3.1, the template architecture constitutes: (1) the memory interface—to transfer data to and from external memory; (2) the shifter—to align the data coming from memory; (3) the prefetch buffer—to store the aligned data; and (4) the two-dimensional array of PEs—to compute partial gradient updates and locally aggregate them. We choose this 2D topology, because it enables the Planner to modularly add or remove PEs as columns or rows. As discussed, this organization also enables an efficient design space exploration by assigning PEs to the worker threads in the rows granularity.

Connectivity and bussing. As Figure 3.1 shows, the number of PEs in each row of the template matches the off-chip bandwidth so that the memory interface can feed all the PEs in a row every cycle, maximizing parallelism. Each row of PEs connects to the memory interface using a pipelined bus, as shown in Figure 3.1. Pipelining the bus is necessary for scalability since the bus is shared by all the rows in the accelerator. In addition to data transfer between external memory and the PEs, connectivity between PEs is required to transfer

intermediate results due to data-dependencies in the DFG. To facilitate the communication, PEs in a single row are connected to their adjacent PEs using bi-directional links and are also connected to the other PEs in the row via a shared bus. A hierarchical tree bus connects the shared bus for different rows. We specialize the interconnect between PEs in the template architecture for communication patterns typical for operations in stochastic gradient descent based learning algorithms. One such example of a common operation is a vector dot product, which involves element-wise multiplication followed by reduction ($\sum$). The result is then typically communicated to all PEs. While the PEs can execute the element-wise multiplication in parallel, the reduction and broadcast operations require significant communication between PEs, which can be a performance bottleneck. In order to alleviate the communication overhead and ensure high utilization of the accelerator's resources, PEs possess three distinct levels of connectivity. Figure 3.1 shows these three levels of connectivity for the template architecture with (n) PEs per row and (m) rows. At the first level, the n adjacent PEs within each row can communicate using bi-directional links. Next, a shared bus connects all of the n PEs within each row. Finally, we use a tree bus to connect the shared bus of m rows of the accelerator. To further aid the reduction operation, each node in the tree bus contains an ALU to perform $\sum$ and $\prod$ operations.

PE design. Figure 3.2 details a PE, the basic unit of the template architecture responsible for executing the operations of the DFG. The rows of PEs within a worker thread exploit fine-grained parallelism in the DFG, enabling the execution of multiple independent operations in parallel. A PE consists of separate buffers for storing training data, model parameters, and intermediate results. This partitioning of buffers is necessary to enable parallel accesses required for DFG operations. The buffers are composed of on-chip SRAMs and the size of each buffer can be configured by the Planner for a given DFG. CoSMIC's Compiler statically generates the schedule of operations for each PE. The PEs execute the scheduled operations using a five stage pipeline, orchestrated by a PE scheduler. The first pipeline stage reads the required data from PE's buffers, adjacent PE links, and shared

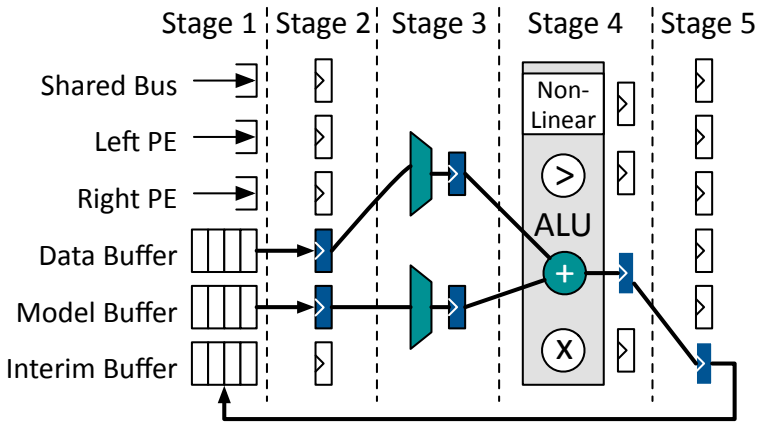


Figure 3.2: Pipelined PE. Black highlights an Add operation (InterimBuffer[i] = DataBuffer[j] + ModelBuffer[k]).

bus links. This data is registered in the second stage. The third stage selects the input operands required by the scheduled operation. The fourth stage executes the scheduled operation using the PE's ALU. For FPGA implementation, the ALU uses DSPs blocks—the hardened on-chip arithmetic unit on the FPGA. The non-linear unit is a look-up table that implements expensive operations like sigmoid, gaussian, divide, and logarithm and is only instantiated in a PE if the Compiler schedules a non-linear operation for that PE. The output of the ALU unit is written back in the fifth and final stage of the PE pipeline. The PEs have a bypass path between the final stage and the ALU stage to forward the result of the previous operation. Figure 3.2 highlights the path taken by an add operation which reads from data and model buffers and writes back to the interim buffer.

Memory interface. Simplicity of the PEs and their highly pipelined design is vital for the efficiency of the accelerator. To further simplify the design, the template architecture prevents the PEs from initiating data requests to the memory. Instead, as illustrated in Figure 3.1, the design harbors a smart memory interface which feeds the PEs according to the schedule generated by the Compiler. This memory interface design is intended to alleviate the overhead of data marshaling, which would have been prohibitive since CoSMIC targets distributed learning with copious amounts of data. However, one issue that arises is that the vectors of data in the off-chip memory do not necessarily align with the rows

of the PEs. This can lead to under-utilization of off-chip bandwidth, which is often a performance bottleneck. To avoid the overhead of padding the data to align with the PEs, we propose to use an on-chip Shifter that aligns input data after fetching it, according to the data map generated by the Compiler. In addition to the Shifter, the memory interface will have a Prefetch Buffer. The size of the training data for each DFG is often large. Hence the time required for external memory access is significant. The Prefetch Buffer enables the accelerator to store the subsequent set of training data for the worker threads, thereby hiding the latency of memory accesses and enabling efficient MIMD execution. The memory interface can also perform broadcast writes to the PEs, as the same model needs to be sent to all the worker threads before they start calculating the new gradient updates.

3.2.2 Multi-Threaded Acceleration

The programmable memory interface plays a significant role in enabling multithreading in the accelerator without imposing significant hardware overhead. It harbors a Memory Schedule queue along with a Thread Index Table that stores thread-specific information as depicted in Figure 3.1. This information includes the memory address of each thread's data sub-partition and the base index of the first allocated PE row to the thread. In addition, each thread has its own dedicated pointer to the Memory Schedule queue. The data transfer schedule is the same for all the threads but it needs to start from different addresses and write to different PEs. The Thread Index Table enables correct and efficient data transfer from memory to all the threads while the schedule is shared. Each row of the table corresponds to one thread. The first field in each row is Mem Addr, which specifies the starting address of each thread's data sub-partition in the off-chip memory. The second field, PE Offset, specifies the index of the first PE of the thread. By walking through these rows, the memory interface controller uses the entries of the Memory Schedule and the Thread Index Table to generate memory accesses for each thread in a round-robin fashion. Each

entry of the schedule stores a Base PE Index, RD/WR bit, Broadcast bit, and Size. The index of the target physical PE is (Base PE Index + PE Offset). The latter term in the addition comes from the Thread Index Table. The memory address is also obtained from the Thread Index Table, which is updated by the size of the transferred data after it finishes. Using this table, the memory interface has the necessary information to transfer each thread's data to its allocated PEs without the need for storing multiple copies of the memory schedule. The RD/WR bit of the memory schedule entry specifies whether the memory access is a read or a write. The Broadcast bit allows a memory read to be sent to all the worker threads via the memory interface bus. This bit is particularly useful when sending model parameters from memory to all worker threads. The Size specifies the size of the data transfer. The Compiler generates the memory schedule according to the Planner-provided architecture and the DFG. The following section discusses the Compiler in detail.

3.3 Evaluation

We evaluate CoSMIC with 10 different machine learning benchmarks using various acceleration platforms, which consist of one FPGA (Xilinx UltraScale+ VU9P) and two P-ASICs. These accelerators are hosted in machines equipped with Intel Xeon E3 v5 processors. We first compare the scalability of the FPGA-accelerated CoSMIC systems to a popular distributed computing platform, Spark [72], while increasing the number of nodes from 4 to 8 to 16. For the scale-out experiments, we used Amazon EC2. We built a local three node system for the in-depth sensitivity studies. We also perform comparison with the distributed GPU (Nvidia K40c) implementation of the benchmarks. Table 3.2 details the specification of these platforms. Lastly, we compare the CoSMIC template architecture with TABLA [45], a single-node FPGA acceleration framework for ML.

Table 3.1: Benchmarks, algorithms, application domains, and datasets.

Name	Algorithm	Domain	Description	# Features	Model Topology	Model Size (KB)	Lines of Code	# Input Vectors	Input Data Size (GB)
mnist	Backpropagation	Image Processing	Handwritten digit pattern recognition	784	784×784×10	2,432	55	60,000	0.4
acoustic		Audio processing	Hierarchical acoustic modeling for speech recognition	351	351×1,000×40	1,527	55	942,626	5.6
stock	Linear Regression	Finance	Stock price prediction	8,000	8,000	31	23	130,503	14.7
texture		Image Processing	Image texture recognition	16,384	16,384	64	23	77,461	17.9
tumor	Logistic Regression	Medical Diagnosis	Tumor classification using gene expression microarray	2,000	2,000	8	22	387,944	10.4
cancer1		Medical Diagnosis	Prostate cancer diagnosis based on the gene expressions	6,033	6,033	24	22	167,219	13.5
movielens	Collaborative Filtering	Recommender System	Movielens recommender system	30,101	301,010	1,176	42	24,404,096	0.6
netflix		Recommender System	Netflix recommender system	73,066	730,660	2,854	42	100,498,287	2.0
face	Support Vector Machine	Computer Vision	Human face detection	1,740	1,740	7	27	678,392	15.9
cancer2		Medical Diagnosis	Cancer diagnosis based on the gene expressions	7,129	7,129	28	27	208,444	20.0

3.3.1 Methodology

Benchmarks and training input datasets. Table 3.1 shows the list of 10 benchmarks—obtained from machine learning literature—that train two different models with each of the following five different algorithms: backpropagation, linear regression, logistic regression, collaborative filtering, and support vector machines. The benchmarks represent various application domains including image processing, audio processing, finance, medical diagnosis, recommendation systems, and computer vision. The mnist and acoustic benchmarks train Multi-Layer Perceptrons (MLPs) for handwritten digit [58, 107] and automatic speech recognition [108], respectively. The stock benchmark trains a linear regression model to predict stock prices using the tick-level data points [109]. The texture benchmark trains another linear regression model for texture recognition [110]. The tumor and cancer1 benchmarks train two different logistic regression models to detect tumors [111] and cancer [112] using the microarray gene expression data. The movielens and netflix benchmarks train recommender systems that employ the collaborative filtering algorithm on Movielens datasets [113, 114] and Netflix Prize Dataset [115]. The face benchmark trains a support vector machine for face recognition [116]. The cancer2 benchmark trains another support vector machine to detect cancer [116]. We train each benchmark for 100 epochs over its dataset. We repeat the experiments 10 times and use the average runtime. In Table 3.1, the "# of Features" column shows the number of elements in each training data vector and the "Model Topology" column denotes the model topology of each benchmark. The "Model Size" column shows the size of the model parameters. The "Lines of Code" column lists the number of lines of code that the programmer writes, which ranges from 22 to 55. Finally, the "# of Input Vectors" and "Input Data Size" columns show the number of the training vectors and the size of the training data. The model parameters for all the benchmarks fit in on-chip memory of the FPGA and the P-ASICs.

Scale-out system specification. Both CoSMIC and Spark systems are deployed on a clus-

ter of machines, which are equipped with the high-performance quad-core Intel Xeon E3 Skylake processors with hyper-threading support that operates at 3.6 GHz. The detailed CPU specification is provided in Table 3.2. The machines run Ubuntu 16.04.1 LTS with the kernel version 4.4.0-47. The machines are connected through a TP-LINK 24-Port gigabit Ethernet switch (TL-SG1024) via TP-Link gigabit Ethernet network interface card (TG'-3468). The switch supports full duplex operation on all ports (2 Gbps per port) and a combined switching capacity of up to 48 Gbps.

Spark. We compare CoSMIC with Spark version 2.1.0. Spark is selected as the point of comparison since it supports efficient in-memory processing for iterative applications. Moreover, Spark provides the MLlib [87] machine learning library. The Spark MLlib library provides the baseline implementation for backpropagation, linear regression, logistic regression, collaborative filtering, and support vector machines [87]. To optimize the performance of MLlib, we build Spark with vectorized OpenBLAS library. For all the Spark results, we use the best-performing combination of machines and threads. The best number of threads is selected for each benchmark individually.

FPGA. As Table 3.2 shows, we use Xilinx Virtex UltraScale+ VU9P for the FPGA experiments. We use Xilinx Vivado 2017.2 to synthesize the generated accelerators at 150MHz. The synthesized accelerators are connected to the external DRAM using the AXI-4 IP.

GPU. For comparison with GPUs, we extend CoSMIC's runtime system to support GPUs since Spark does not. The alternative would have been integrating GPUs with Spark, which

Table 3.2: CPU, GPU, FPGA, and P-ASICs.

	CPU	GPU		FPGA		P-ASIC	P-ASIC
Chip	Xeon E3-1275 v5	Tesla K40c	Chip	UltraScale+ VU9P	Chip	F	G
Cores	4	2,880	DSP Slices	6,840	PEs	768	2,880
Memory	32 GB	12 GB	BRAM	44,280 KB	Area (mm^2)	29	105
TDP	80 W	235 W	TDP	42 W	Power	11 W	37 W
Frequency	3.6 GHz	875 MHz	LUTs	1,182 K	Frequency	1 GHz	1 GHz
Technology	14 nm	28 nm	Flip Flops	2,364 K	Technology	45 nm	45 nm

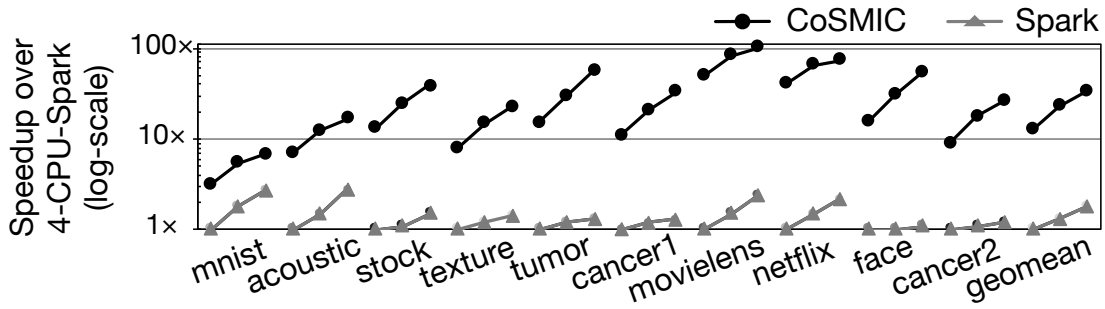
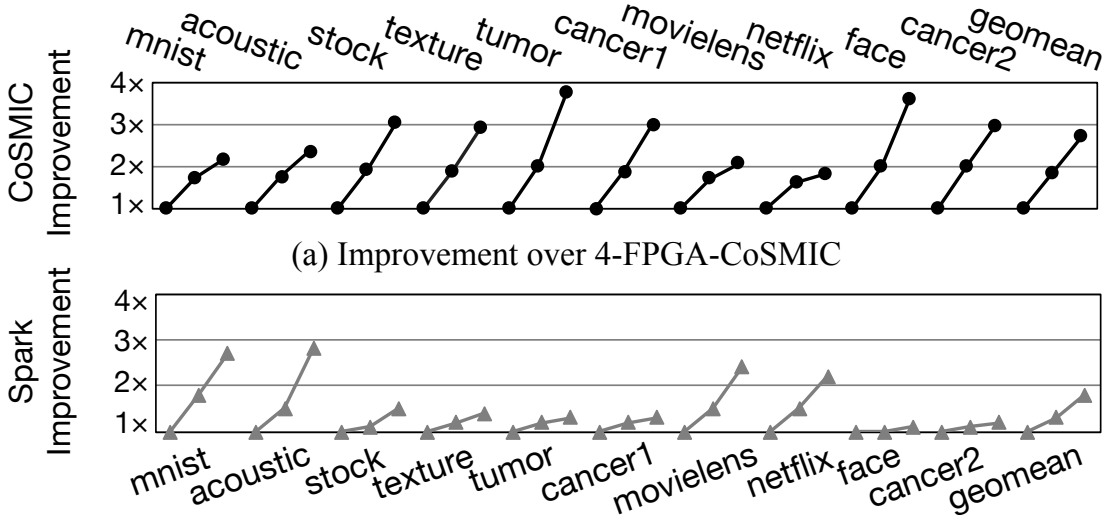


Figure 3.3: Speedup over Spark as the number of nodes increases from 4 to 8 to 16. Baseline: Spark system with 4 nodes (4-CPU-Spark).



(a) Improvement over 4-FPGA-CoSMIC

(b) Improvement over 4-CPU-Spark

Figure 3.4: Scalability comparison of CoSMIC and Spark as the number of nodes increases from 4 to 8 to 16.

is on its own a line of ongoing research [117, 118, 119, 120]. As such, we build a GPU-accelerated CoSMIC system. We had three Nvidia Tesla K40 GPUs at our disposal, which are used for this comparison (see Table 3.2 for hardware specification). For the GPU experiments, we developed highly optimized CUDA implementations using well-known libraries, including LibSVM-GPU [121] and Caffe2+cuDNN [122], as well as source code from related works [43, 45]. In all cases, we used the latest versions of each library (e.g., cuBLAS v8.0 [123] and cuDNN v7.0 [124]). We use WattsUp [125] to measure the system power following the same methodology in the prior work [126].

P-ASICs. We use Synopsys Design Compiler (L-2016.03-SP5) and TSMC 45-nm high-Vt standard cell libraries to synthesize the CoSMIC-generated architectures and obtain the area,

frequency, and power results. We used CoSMIC to generate two P-ASIC designs: one with the PE count and off-chip bandwidth that match those of the FPGAs (P-ASIC-F), the other that match those of the GPUs (P-ASIC-G). Table 3.2 provides the details of these P-ASICs. We combine the system-level measurements with the synthesis and simulation/estimation results to evaluate these P-ASICs.

3.3.2 Experimental Results

Performance comparison. Figure 3.3 shows the result of performance comparison between CoSMIC and Spark using three system configurations: 4-Node, 8-Node, and 16-Node. The baseline is a 4-Node Spark system, referred to as 4-CPU-Spark. On average, the 4-FPGA-, 8-FPGA-, 16-FPGA-CoSMIC configurations deliver 12.6×, 23.1×, and 33.8× higher performance, respectively. Whereas, increasing the number of nodes with Spark from 4 to 16 only yields 1.8× performance improvement. The performance does not scale linearly as the number of nodes increases due to system management overhead in networking and aggregation. The performance gains for different benchmarks depend on their model topology, parallelism, and memory footprint. For example, movielens (collaborative filtering) sees the highest speedup (100.7×) since its DFG is significantly parallel that allows CoSMIC to utilize the FPGAs resources for higher performance. On the contrary, mnist and acoustic (backpropagation) achieve relatively smaller speedup (6.8× and 16.5×) since these benchmarks require significant on-chip communication, which bottlenecks performance. These results show that CoSMIC's full-stack approach, which comes with our multithreaded accelerators, is highly effective for the scale-out acceleration of these ML applications. Furthermore, these results show that CoSMIC better utilizes the added resources and is more scalable as the number of nodes increases.

Scalability. To better compare the scalability of the two systems, Figure 3.4 shows the performance improvement over each system's own 4-Node configuration. Figure 3.4(a) shows the improvement with CoSMIC when the 4-FPGA-CoSMIC is the baseline and Figure 3.4(b)

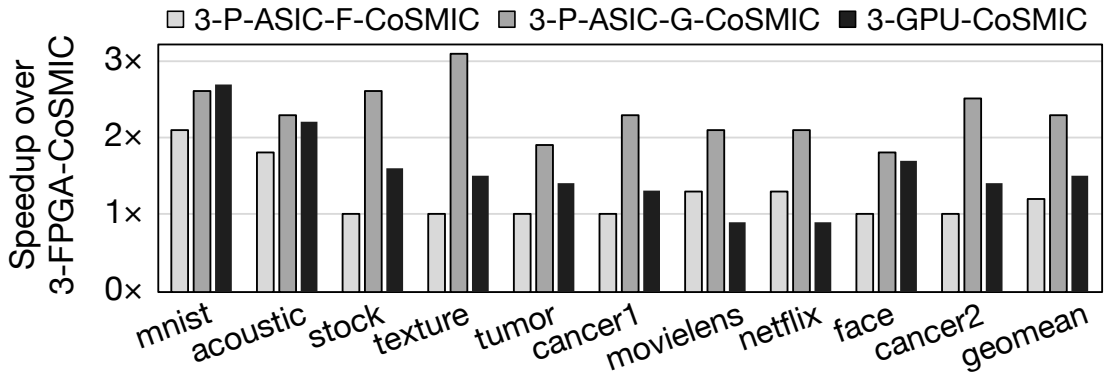


Figure 3.5: System-wide speedup over 3-FPGA-CoSMIC.

shows the improvement with Spark when 4-CPU-Spark is the baseline. On average, CoSMIC performs 1.8× and 2.7× faster when the system is scaled up to 8 and 16 nodes, respectively. As a point of reference and comparison, Spark shows 1.3× and 1.8× speedup for the same increase in the number of nodes. The results from Figure 3.3 and Figure 3.4 show that CoSMIC scales well and better than Spark as the number of nodes increases. The improvement gap between Spark and CoSMIC is larger for the benchmarks that have higher ratio of communication to computation in the runtime (stock, texture, tumor, cancer1, face, and cancer2). For the other benchmarks, CoSMIC scales less steeply in comparison to Spark. These benchmarks are compute-bound and therefore acceleration is effective and adding accelerators reduces the computation time in the baseline 4-Node configuration. Since Spark does not utilize the accelerators, it benefits more from the added nodes as they bring in the necessary compute power that was missing in the 4-Node configuration. Therefore, adding more nodes helps but it is more effective for Spark. Nonetheless, as Figure 3.3 illustrates, CoSMIC significantly outperforms Spark across all the benchmarks. These results confirm that the specialization of the system software has been effective in enabling acceleration at scale.

Comparison of different acceleration platforms. Figure 3.5 compares the benefits of CoSMIC with FPGAs and P-ASICs to GPUs. The results are obtained from our three-node system configuration and the baseline is the 3-FPGA-CoSMIC. On average, the 3-P-ASIC-F-CoSMIC, 3-P-ASIC-G-CoSMIC, and 3-GPU-CoSMIC systems provide average 1.2×, 2.3×,

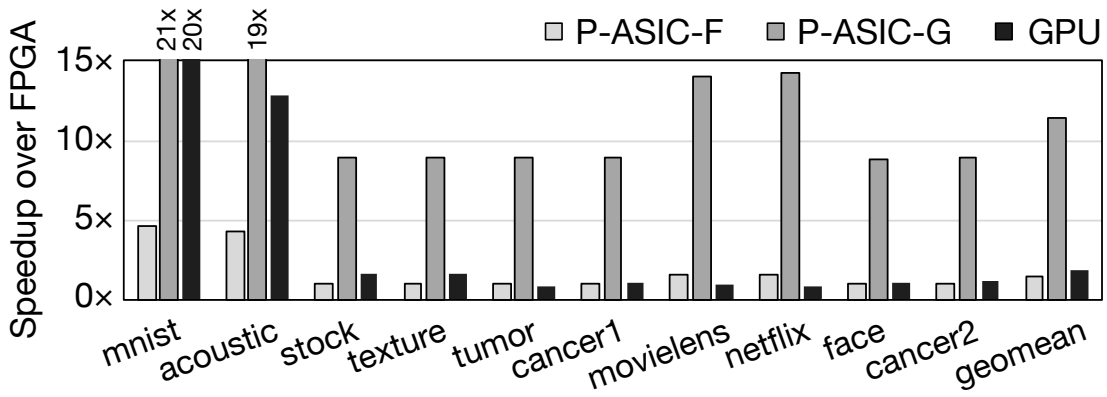


Figure 3.6: Computation speedup over FPGA.

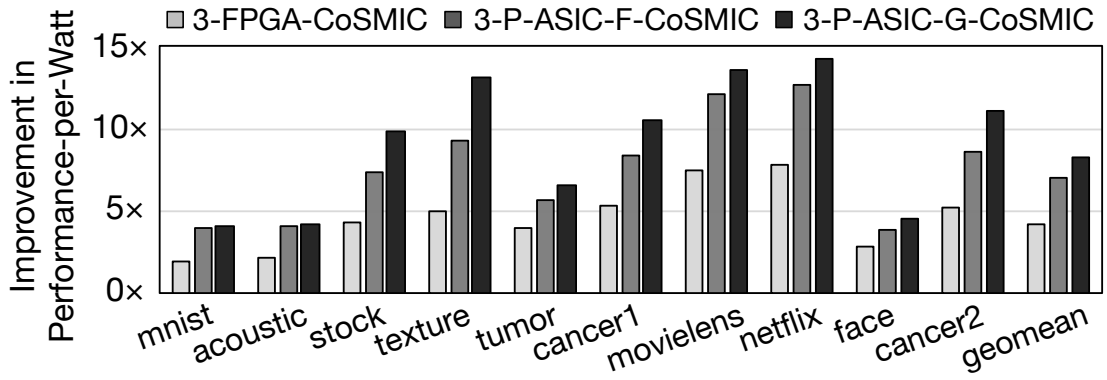
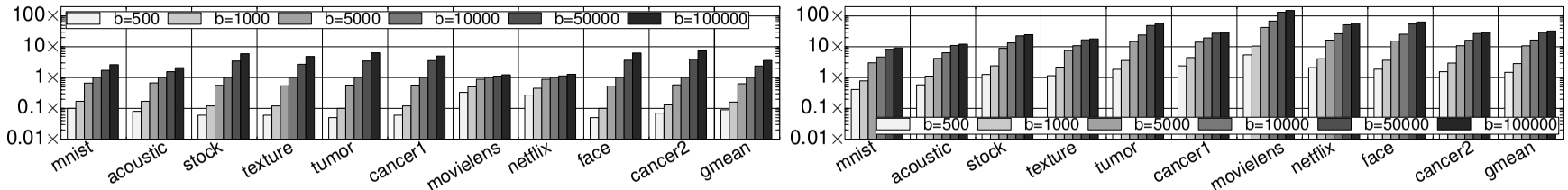


Figure 3.7: Performance-per-Watt, baseline: 3-GPU system.



(a) Spark performance as the mini-batch size increases. (Log-scale)

(b) CoSMIC performance as the mini-batch size increases. (Log-scale)

Figure 3.8: Performance vs. mini-batch size as it is swept from 500 to 100,000; baseline: 3-node Spark when the mini-batch size is 10,000.

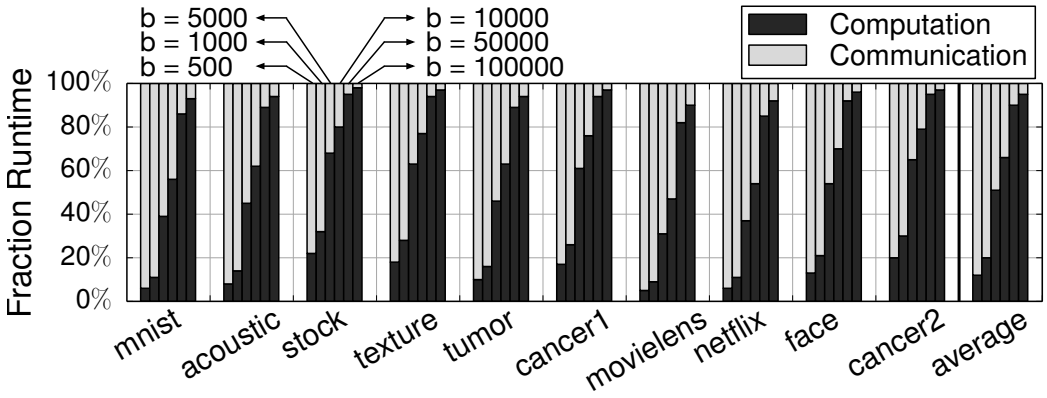


Figure 3.9: Fraction of 3-FPGA-CoSMIC runtime.

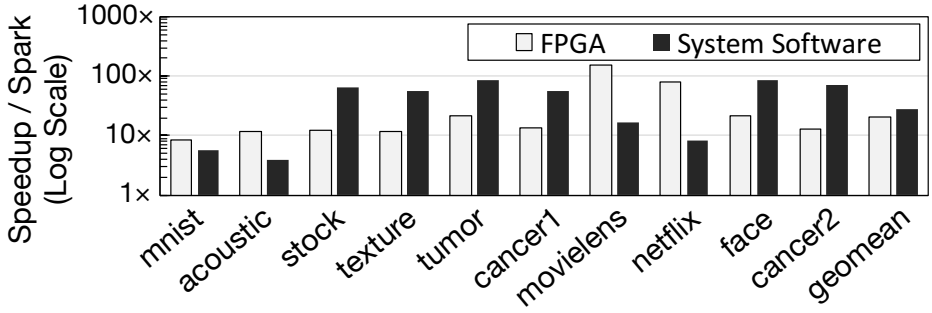


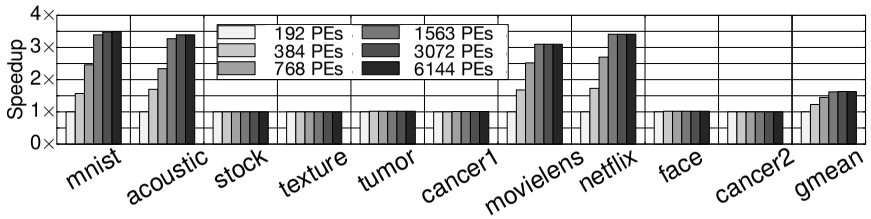
Figure 3.10: Speedup breakdown between FPGAs and system software (aggregation, networking, and management) for 3-FPGA-CoSMIC.

and 1.5× higher performance than the 3-FPGA-CoSMIC system, respectively. Although as expected P-ASICs and the GPU outperform the FPGA, the benefits are relatively modest. To understand this trend, Figure 3.6 shows the improvement in compute time without considering the system software. On average, P-ASIC-F, P-ASIC-G, and GPU perform 1.5×, 11.4×, and 1.9× faster than FPGA, respectively. Except for mnist and acoustic benchmarks, which use the backpropagation algorithm, the benefits from P-ASIC-F and GPU are not overwhelming. GPU provides higher speedup on two specific benchmarks (20.3× for mnist and 12.8× for acoustic) as the dominant part of their computation is relatively large matrix-matrix multiplication that GPUs can compute very efficiently. P-ASIC-F offers the same number of PEs and bandwidth compared to the FPGA but at higher frequency. These results show that just improvement in frequency does not translate to proportional speedup as long as the bandwidth remains unchanged. These results also show that the coalescence of CoSMIC's Planner, Compiler, and multi-threaded accelerator design has been effective

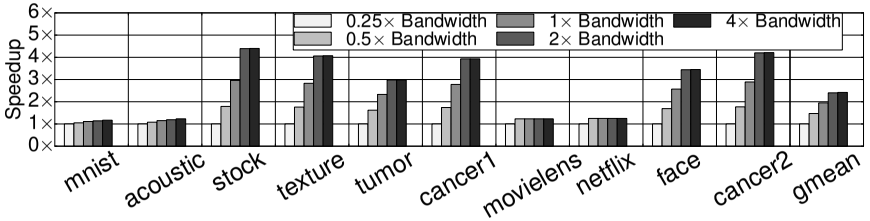
in exploiting the FPGA resources. Across all benchmarks, P-ASIC-G shows significantly higher improvement as this design point combines more PEs, higher frequency, and higher bandwidth. The PE count and bandwidth of P-ASIC-G matches the GPU and its frequency is higher than the FPGA. However, as Figure 3.5 illustrates, even in the case of P-ASIC-G, the computation speedup does not translate to proportional system-wide improvement. These results confirm the importance of system software and CoSMIC-like full-stack approaches, as accelerators gain popularity.

The speedup of 3-GPU-CoSMIC comes from the GPU's higher frequency as well as massive parallelism; however, it also comes at an expense of higher power dissipation. Figure 3.7 highlights this power-performance tradeoff by depicting the improvement in Performance-per-Watt when comparing the FPGA- and P-ASIC-accelerated systems to the GPU-based system. The 3-FPGA-CoSMIC, 3-PASIC-F-CoSMIC, and 3-PASIC-G-CoSMIC systems achieve on average 4.2×, 6.9×, and 8.2× higher Performance-per-Watt than 3-GPU-CoSMIC, respectively. These results show that when the power-efficiency is the main concern, FPGAs or P-ASICs will be more desirable acceleration platforms than GPUs although GPUs provide higher performance than FPGAs and one of the P-ASICs, namely P-ASIC-F. Moreover, although P-ASICs provide both higher performance and power-efficiency, they impose a significant design and manufacturing cost. CoSMIC's template approach reduces the design time and cost as it offers a way to generate accelerator code. However, the cost of manufacturing may tip the scale towards FPGAs as they also offer significant benefits in both performance and power efficiency.

Sensitivity to mini-batch size. We use 10,000 as the default mini-batch size as used in the machine learning literature [127, 128, 129]. However, the optimal mini-batch size depends on several variables such as model, datasets, and training iterations. Larger mini-batch size reduces the rate of aggregation, which reduces the inter-node communication, leading to higher performance. Figure 3.9 illustrates this effect by segregating the fraction of runtime spent in computation and communication as the number of mini-batch size



(a) Speedup of CoSMIC accelerator with increasing number of PEs
(Baseline: CoSMIC accelerator with 192 PEs)



(b) Speedup of CoSMIC accelerator as off-chip bandwidth changes
(Baseline: CoSMIC accelerator using 25% of UltraScale+ bandwidth)

Figure 3.11: Speedup comparison with varying number of PEs and memory bandwidth for CoSMIC accelerators.

Figure 3.12: Design space exploration; Tx×Ry, x represents the number of threads and y represents the number of rows; baseline: T1×R1.

increases from b=500 to b=100,000 in the three-node runtime. On average, the computation with the mini-batch size 500 takes 12% of runtime but this increases to 95% when the mini-batch size is 100,000. However, reducing the aggregation rate can adversely affect training convergence [127, 128, 130, 129, 131]. To study the effect of mini-batch size on Spark and CoSMIC, we sweep the mini-batch size from 500 to 100,000 for three-node system configuration. Figure 3.8(a) and Figure 3.8(b) present the result of this sweep. For both figures, the baseline is the three-node Spark when mini-batch size is 10,000, our default setting. Comparing Figure 3.8(a) and Figure 3.8(b) shows that 3-FPGA-CoSMIC is faster across all combinations of benchmarks and mini-batch sizes. On average, with the same mini-batch size of b=500, CoSMIC is 16.8× faster. When the mini-batch size increases to b=100,000, CoSMIC is 9.1× faster. As the mini-batch size increases, Spark's overheads diminish. Nevertheless, CoSMIC outperforms Spark.

Sources of speedup. Figure 3.10 teases apart the benefits of FPGA acceleration from the benefits of the specialized system software over the three-node Spark. On average, the three FPGAs provide 20.7× speedup and the specialized system software–which also includes the aggregation part of the computation–is 28.4× faster than Spark's system software. As we discuss below, six of the benchmarks are more sensitive to data transfer and thus gain more benefits from the specialized system software compared to the benefits from FPGA. These benchmarks specifically benefit from the system software's task assignment that utilizes CPUs for both networking and aggregation of partial results from other nodes, thereby avoiding extra data transfer to the FPGAs. Nonetheless, all benchmarks gain from both FPGAs acceleration and specializing the system software.

Sensitivity to FPGA resources and bandwidth. CoSMIC can reshape and customize the template to match the resources of the target FPGAs or P-ASICs. The two main resources that affect performance are the number of PEs and the off-chip memory bandwidth. However, the DFG of the learning algorithm determines which resource is dominant. To study the interplay of algorithms and resources, we use a performance estimation tool that is val-

Table 3.3: Number of threads and FPGA resource utilization.

Name	# Threads per FPGA	LUTs (Total: 1,182,240)		Flip Flops (Total: 2,364,480)		BRAM (Bytes) (Total: 9720 KB)		DSP Slices (Total: 6840)	
		Used	Util	Used	Util	Used	Util	Used	Util
mnist	2	851,276	72.0%	772,029	32.7%	8,640	88.9%	4,070	59.5%
acoustic	2	851,276	72.0%	772,029	32.7%	8,128	83.6%	4,070	59.5%
stock	8	278,838	23.6%	249,907	10.6%	8,640	88.9%	1,320	19.3%
texture	1	283,535	24.0%	257,005	10.9%	8,640	88.9%	1,355	19.8%
tumor	4	281,522	23.8%	253,963	10.7%	8,640	88.9%	1,340	19.6%
cancer1	2	282,864	23.9%	255,991	10.8%	8,640	88.9%	1,350	19.7%
movielens	2	851,276	72.0%	772,029	32.7%	8,128	83.6%	4,070	59.5%
netflix	1	851,947	72.1%	773,043	32.7%	8,128	83.6%	4,075	59.6%
face	4	281,522	23.8%	253,963	10.7%	8,640	88.9%	1,340	19.6%
cancer2	2	282,864	23.9%	255,991	10.8%	8,640	88.9%	1,350	19.7%

idated against the hardware. Figure 3.11(a) illustrates the performance changes when the number of PEs varies from 192 to 6144 for a CoSMIC accelerator. The benchmarks that use the backpropagation (mnist and acoustic) and collaborative filtering algorithms (movielens and netflix) algorithms show performance benefits as the number of PEs increases, since they are compute-bound. The rest of the benchmarks–linear regression, logistic regression, and support vector machines do not see any performance gains when the number of DSPs increases. Although these benchmarks are offered more PEs, the limited bandwidth curtails their performance. Figure 3.11(b), which sweeps bandwidth, suggests the same categorization (bandwidth-bound vs. compute-bound) for our algorithms. These results show that a single fixed design is not the most optimal for all the algorithms. Therefore, there is a need for template architectures and solutions, such as CoSMIC, that customize the accelerator design according to the algorithm. These results also suggest that modern accelerators need to strike a balance on allocating resource to off-chip communication and on-chip computation to maximize benefits for all benchmarks. Nonetheless, CoSMIC finds an optimal accelerator design considering both compute and bandwidth resources available on the FPGA.

Design space exploration. The Planner determines the number of PEs per thread and the number of threads in the accelerator. The Planner allocates PEs to each thread at the granularity of one row. This allocation strategy limits the design space that the Planner explores

to find the optimal number of threads and rows-per-thread. In the case of UltraScale+ VU9P FPGA, the maximum number of possible design points is 27. Also, recall that the number of threads is also limited by the size of the model and not all the design points are possible. Figure 3.12 illustrates the result of this design space exploration for four different benchmarks. The performance of each design point is normalized to the design point which runs 1 thread using 1 row (T1xR1) of PEs. We sweep the number of rows from 1 to 48, which is the maximum number of rows in UltraScale+ while the maximum number of threads varies for every benchmark. The optimal design points are highlighted with a concentric circle in the graphs. Benchmarks mnist and movielens see the highest speedup when they use all the 48 rows since they are compute-bound. In contrast, the performance for stock and tumor saturates beyond 16 rows. This result is commensurate with Figure 3.11(a), which shows that mnist and movielens benefit significantly with an increase in the FPGA's computational resources (PEs), while stock and tumor do not. The rest of the benchmarks show trends similar to the ones in Figure 3.12. Further, the figure shows that for a fixed number of PE rows, increasing the number of threads improves performance, which confirms the importance of multi-threading. Table 3.3 shows the resource utilization and the optimal number of threads-per-FPGA for all the benchmarks corresponding to the optimal design point chosen by the Planner. The resource utilization is highest for benchmarks that are compute-bound and lowest for the benchmarks that are bandwidth-bound. Moreover, the results show the benefits of our template-based approach that enables optimal utilization of the limited resources in the FPGA's reconfigurable fabric.

Comparison with TABLA. Prior work in TABLA [45] has explored single-node acceleration using a low-power FPGA (Zynq ZC702 with 220 DSPs). Our work, on the other hand, explores scale-out acceleration using modern high-power FPGAs (UltraScale+ with 6,840 DSPs). To provide a head-to-head comparison, we use the open-source TABLA framework [132] to generate accelerators for UltraScale+. We modify the templates for UltraScale+ and perform design space exploration to present the best results with TABLA.

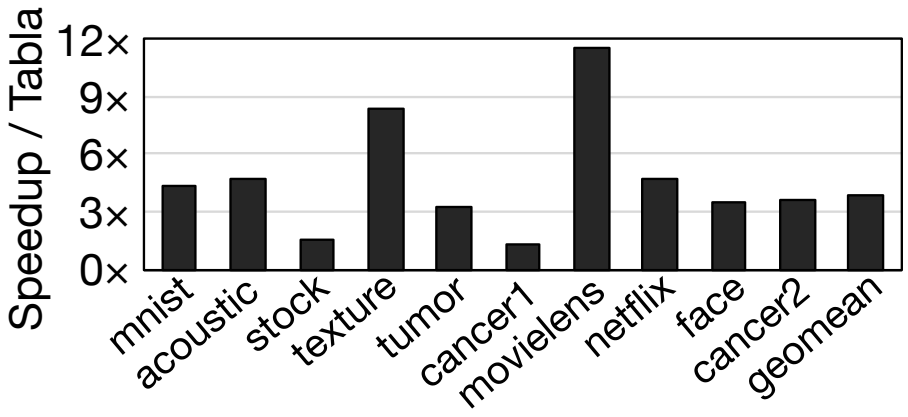


Figure 3.13: Speedup of CoSMIC's template architecture over TABLA's.

Figure 3.13 shows the speedup of CoSMIC compared to TABLA on UltraScale+ when using the same number of PEs. On average, CoSMIC performs 3.9× faster than TABLA. While both CoSMIC and TABLA use the same number of FPGA compute resources, the gap in performance shows that CoSMIC uses the compute resources more efficiently. The bottleneck for performance in TABLA is the communication of intermediate results due to data dependencies. As the number of DSPs in the TABLA architecture grows, the communication overhead grows significantly. To reduce the communication overhead, CoSMIC architecture uses a scalable tree-bus across rows of our 2-D PE architecture, and a bidirectional link between columns of PEs. Moreover, TABLA's compiler does not consider the overhead of data communication, which is particularly important when the number of PEs is large. CoSMIC compiler (See our MICRO paper for details) maps the operations of the learning algorithm according to the location of data in order to reduce communication overhead. The combination of CoSMIC's scalable architecture, along with compiler optimization ensures that the FPGA's computational resources are used effectively.

3.4 Related Work

Multi-node accelerators for machine learning. DaDianNao [36] provides a multi-chip ASIC accelerator for DNNs. Other works use multiple FPGAs for accelerating one specific

task [133, 134, 135]. Farabet et al. [133] and Donninger et al. [134] use multiple FPGAs to accelerate DNNs [133] and a chess game [134], respectively. Walters et al. [135] propose a multi-FPGA accelerator for the Hidden Markov Models [135]. Putnam et al. [42] provide an FPGA fabric for accelerating Bing's ranking algorithms [42]. Microsoft [75] also provides an infrastructure for deploying FPGAs in datacenters, which is also used for the inference phase of DNNs. This release does not deal with training nor does it offer a framework for programming. CoSMIC provides the necessary framework to utilize and program such an infrastructure [75] for machine learning algorithms without involving programmers in hardware design. Recently, Microsoft also unveiled Brainwave [136] that uses multiple FPGAs for DNN inference. In contrast, CoSMIC is a full stack to accelerate training at scale. Google's TPU [137] is a systolic array for acceleration of matrix multiplication, which is prevalent operation in ML. TPU is also programmable from Tensorflow [138] that recently supports distributed execution. In contrast, CoSMIC enables the use of FPGAs for scale-out acceleration and comes with its own template architecture.

Template-based acceleration. TABLA [45] is a single-node accelerator generator for machine learning, which also uses a template-based architecture. As discussed in Section 3.3, TABLA, developed for a low-power FPGA (Zynq), does not effectively utilize the resources of a modern server-scale FPGA (UltraScale+). Furthermore, TABLA generates single-node FPGA accelerators which are inherently limited by the fine-grained parallelism available in the single-thread of stochastic gradient descent. In contrast, the CoSMIC framework not only generates *scalable* accelerators for distributed systems using a novel *multi-threaded* template architecture, but also provides the necessary system software stack for scale-out acceleration. Moreover, the compilation algorithm of this work differs from TABLA. Our algorithm reduces the data communication by mapping data first. In contrast, TABLA's algorithm maps operations first to reduce the single-threaded latency. Additionally, our algorithm optimizes the mapping of operation to the FPGA's resources according to the location of data to avoid data marshaling. DNNWEAVER [24] is another template-based

accelerator generator that only generates accelerators for prediction with Deep Neural Networks (DNNs). DNNWEAVER does not deal with training, multiple FPGAs, or algorithms besides DNNs. Cheng, et al. [139] propose predesigned data flow templates as the intermediate point for HLS from general C/C++ workloads. LINQits [140] provides a template architecture for accelerating database queries. The last two works [139, 140] do not focus on learning algorithms nor do they deal with scale-out systems.

Single-node accelerators for machine learning. There is a large body of work on single-node accelerator design for ML [88, 89, 90, 91, 92, 35, 43, 70, 69, 46, 71, 44, 93, 7, 10, 94, 95, 96, 97, 98, 99, 100, 101, 102, 98, 99, 103, 104, 105, 106, 37]. These works mostly focus on accelerating one or a fixed number of learning algorithms. CoSMIC, on the other hand, is a full stack that targets scale-out acceleration of learning.

HLS for FPGAs. Many related works (e.g., [139, 146, 147, 37]) explore HLS for FPGAs. HLS targets general applications while CoSMIC focuses specifically on machine learning. Therefore, HLS does not leverage any domain-specific knowledge or algorithmic insights. Using algorithmic commonalities for a range of machine learning algorithms is fundamental to our work and enables further benefits from hardware acceleration. Acceleration with HLS still requires hardware expertise. For instance, DNNWEAVER [24] reports that hardware design to optimize a Vivado HLS implementation of a deep neural network for FPGA took one month. The resulting implementation was an order of magnitude slower than a template-based accelerator for the same FPGA. A more recent work [139] uses dataflow templates as intermediate compilation target for C/C++ programs and delivers 9× higher performance than state-of-the-art HLS tools. CoSMIC takes a template-based approach that is driven by the theory of machine learning and targets distributed FPGA acceleration of training from a high-level domain-specific language.

System software for distributed FPGA acceleration. Another inspiring work [148] provides the mechanisms to integrate predesigned FPGA accelerator with Spark [72]. Melia [149] uses Altera's OpenCL-based HLS to offer a MapReduce-based framework for utilizing FP-

GAs in distributed systems. Another work [150] provides the framework for using Xilinx Vivado HLS tool for MapReduce [151] applications. CoSMIC does not rely on predeveloped FPGA accelerators or HLS for distributed FPGA acceleration, or generic system software.

3.5 Conclusion

While accelerators gain traction, their integration in the system stack is not well understood. CoSMIC takes an initial step toward such an integration for an important class of applications while providing generality and a high-level programming interface. The evaluations confirm that a full-stack approach is necessary and just designing efficient accelerators does not yield proportional benefits without a co-design of the entire system stack. The traditional approaches of profiling and offloading hot-regions of code lack the flexibility to support ever-changing algorithms and the emerging scale and heterogeneity in the systems. It is clear that a full-stack design is non-trivial but deeply understanding algorithmic properties of the application domain can significantly facilitate such approaches. CoSMIC takes advantage of the algorithmic understanding to simplify the layers of its stack by specializing them and offers a cohesive hardware-software solution. The encouraging results show that this paradigm is effective but the multifaceted nature of the cross-stack approach promises an exciting yet challenging road ahead.

CHAPTER 4
USING ALGORITHMIC INSIGHTS TO ENABLE DNN ACCELERATION AT THE EDGE

The final thrust of the thesis aims to exploit algorithmic insights to push the envelope for performance and energy efficiency of DNN accelerators. Enabling deep learning in mission-critial energy constrained edge systems, such as drones and wearables, requires a level of efficiency that can only be achieved by understanding and leveraging algorithmic properties of deep learning. As such, thesis chapter shows the ability to achieve performance comparable to a server-grade GPU within the power budget of an edge device.

4.1 Bit-level Dynamically Composability for Accelerating Deep Neural Networks

This thesis chapter first discusses the Bit Fusion architecture based on our ISCA paper. To achieve unparalleled efficiency this chapter builds upon the algorithmic insight, as reported by very recent machine learning literature [152, 153, 154, 155, 156], that bitwidth of operations in Deep Neural Networks (DNNs) can be reduced without compromising their classification accuracy, which is necessary for mission-critical systems. However, to prevent loss of accuracy, the bitwidth varies significantly across DNNs and it may even be adjusted for each layer individually. Thus, a fixed-bitwidth accelerator would either offer limited benefits to accommodate the worst-case bitwidth requirements, or inevitably lead to a degradation in final accuracy. To alleviate these deficiencies, this paper introduces *dynamic bit-level composability* as a new dimension in the design of DNN accelerators. Our work explores this dimension by designing Bit Fusion, a bit-flexible accelerator, that constitutes an array of bit-level processing elements that dynamically fuse to match the bitwidth of individual DNN layers. This flexibility in the architecture enables hyper-efficiency by minimizing the computation and the communication at the finest granularity possible, with

no loss in accuracy. Using bit-level flexibility, Bit Fusion almost matches the performance of a server-grade Nvidia Titan Xp GPU running with 8-bit vales and a TDP of 250 Watts, while consuming just 895 milliWatts of power.

4.1.1 Introduction

Advances in high-performance computer architecture design has been a major driver for the rapid evolution of Deep Neural Networks (DNN). Due to their insatiable demand for compute power, naturally, both the research community [46, 47, 157, 69, 89, 158, 36, 35, 43, 44, 159, 70, 71, 67, 90, 160, 161, 50, 66, 68, 37, 24, 162, 64, 65, 163, 164, 165] as well the industry [136, 137, 166] have turned to accelerators to accommodate modern DNN computation. However, the algorithmic properties of DNNs have not fully been utilized to push the envelope on their acceleration efficiency and performance.

To that end, we leverage the following three algorithmic properties of DNNs to introduce a novel acceleration architecture, called Bit Fusion. (1) DNNs are mostly a collection of massively parallel multiply-adds. (2) The bitwidth of these operations can be reduced with no loss in accuracy [152, 153, 154, 155, 156]. (3) However, to preserve accuracy, the bitwidth varies significantly across DNNs and may even be adjusted for each layer individually. Thus, a fixed-bitwidth accelerator design would either yield limited benefits to accommodate the worst-case bitwidth requirements, or inevitably lead to a degradation in final accuracy. To alleviate these deficiencies, Bit Fusion introduces the concept of runtime bit-level fusion/decomposition as a new dimension in the design of DNN accelerators. We explore this dimension by designing a bit-flexible accelerator, which comprises an array of processing engines that fuse at the bit-level granularity to match the bitwidth of the individual DNN layers.

The bit-level flexibility in the architecture enables minimizing the computation and the communication at the finest granularity possible with no loss in accuracy. As such, the following three insights both motivate and guide Bit Fusion.

DNN	AlexNet	Cifar-10	LSTM	LeNet-5	ResNet-18	RNN	SVHN	VGG-7
% Multiply-Add	99.8	99.8	99.9	99.4	99.9	99.9	99.8	99.5

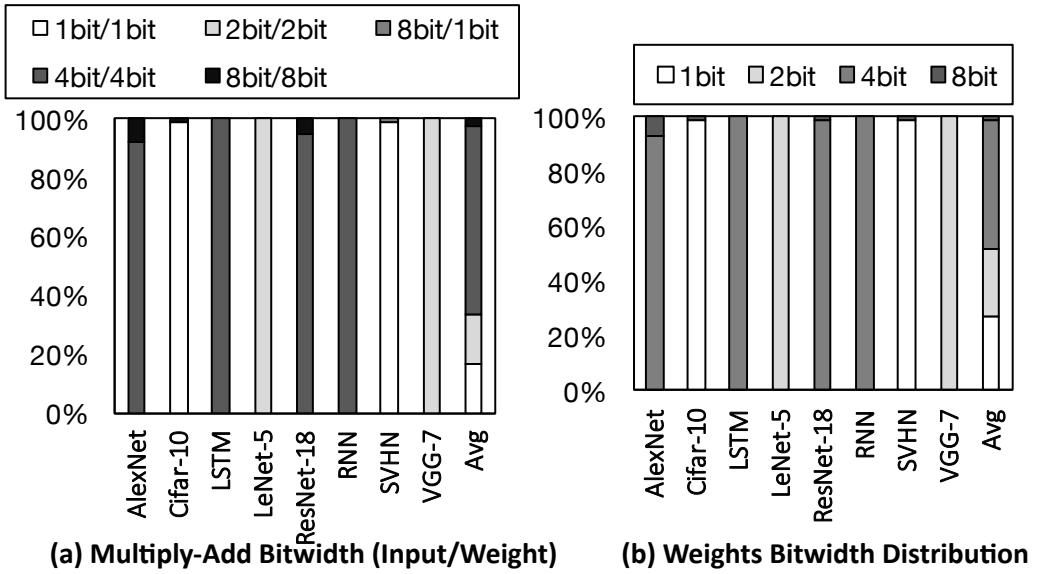


Figure 4.1: Bitwidth variation across real-world DNNs.

First, the number of bit-level operations required for the multiply operator is proportional to the product of the operands' bitwidths and scales linearly for the addition operator. Therefore, matching the bitwidth of the multiply-add units to the reduced bitwidth of the DNN layers, almost quadratically reduces the bit-level computations. This strategy will significantly affect the acceleration since the large majority of DNN operations ($> 99\%$) are multiply-adds as shown in the table included in Figure 4.1. For instance, each single image classification with AlexNet [156] requires a total of 2682 million operations, of which 99.86% (2678 million) are multiply-adds. To this end, the compute units of Bit Fusion can dynamically fuse or decompose to match the bitwidth of each individual multiply-add operand without requiring the operands to be encoded in the same bitwidth.

Second, energy consumption for DNN acceleration is usually dominated by data accesses to on-chip storage and off-chip memory [157, 46, 47]. Therefore, Bit Fusion comes with encoding and memory access logic that stores and retrieves the values in the lowest required bitwidth. This logic reduces the overall number of bits read or written to on-chip and off-chip memory, proportionally reducing the energy dissipation of memory accesses.

Furthermore, this strategy increases the effective on-chip storage capacity.

Third, Bit Fusion builds upon the extensive prior work that shows DNNs can operate with reduced bitwidth without degradation in classification accuracy [152, 155, 154, 153, 167, 89]. This opportunity exists across different classes of real-world DNNs, as shown in Figure 4.1. One category is Convolutional Neural Networks (CNNs) that usually use convolution and pooling layers followed by a stack of fully-connected layers. AlexNet, Cifar-10, LeNet-5, ResNet-18, SVHN, and VGG-7 in Figure 4.1 belong to this category. Recurrent Neural Networks (RNN) are another sub-class of DNNs that use recurrent layers including Long Short Term Memory (LSTM) and vanilla RNN layers to extract *temporal* features from time-varying data. The RNN and LSTM benchmark DNNs in Figure 4.1 represent these categories. Furthermore, as the table in Figure 4.1 shows, most operations in DNNs ($> 99\%$), regardless of their categories, are multiply-adds. As Figure 4.1(a) illustrates, on average, 97.3% of multiply-adds require four or fewer bits and even in some DNNs a large fraction of the operations can be done with bitwidth equal to one. More interestingly, the bitwidths vary within and across DNNs to guarantee no loss of accuracy. Such a variation is not limited to the intermediate operands and exists in trained weights as illustrated in Figure 4.1(b). To exploit this property, a programmable accelerator needs to offer bit-level flexibility at runtime, which leads us to Bit Fusion.

To harvest the aforementioned opportunities, this thesis chapter makes the following contributions and realizes a new dimension in the design of DNN accelerators.

1. **Dynamic bit-level fusion and decomposition.** This chapter introduces and explores the dimension of bit-level flexible DNN accelerator architectures, Bit Fusion, that dynamically matches bit-level composable processing engines to the varying bitwidths required by DNN layers. By offering this flexibility, Bit Fusion aims to minimize the computation and communication required by a DNN at the bit granularity on a per layer basis.
2. **Microarchitecture design for bit-level composability.** To explore Bit Fusion, we design and implement a DNN accelerator using a novel bit-flexible computation unit,

called BitBricks. The accelerator supports both feed-forward (CNN) and recurrent (LSTM and RNN) layers. A 2D array of BitBricks constructs a fusible processing engine that can perform the DNN computation at various bitwidths. The microarchitecture also comes with a storage logic that allows feeding the BitBricks with different bitwidth operands.

3. **Hardware-software abstractions for bit-flexible acceleration.** To enable DNN applications to take advantage of these unique bit-level fusion capabilities, we propose a block-structured instruction set architecture, called Fusion-ISA. To amortize the cost of programmability, Fusion-ISA expresses operations of DNN layers as bit-flexible instruction blocks with iterative semantics.

These three contributions define the novel architecture of Bit Fusion, a possible microarchitecture implementation, and the hardware-software abstractions to offer bit-level flexibility. Other complementary and inspiring works have explored bit serial computation [89, 158] without exploring the fusion dimension. In contrast, Bit Fusion *spatially* fuses a group of BitBricks together, to collectively execute operations at different bitwidths. Using eight real-world feed-forward and recurrent real-world DNNs, we evaluate the benefits of Bit Fusion. We implemented the proposed microarchitecture in Verilog and synthesized in 45 nm technology. Using the synthesis results and cycle accurate simulation, we compare the benefits of Bit Fusion to two state-of-the-art DNN accelerators, Eyeriss [46] and Stripes [89]. The latter is an optimized bit-serial architecture. In the same area, frequency, and technology node, Bit Fusion offers 3.9× speedup and 5.1× energy savings over Eyeriss. Compared to Stripes [89], Bit Fusion provides 2.6× speedup and 3.9× energy reduction at 45 nm node when Bit Fusion area and frequency are set to those of Stripes. Scaling to GPU technology node of 16 nm, Bit Fusion provides a 16× speedup over the Jetson TX2 mobile GPU. Further, Bit Fusion almost matches the performance of a 250-Watt Titan Xp, which uses 8-bit vector instructions, while Bit Fusion merely consumes 895 milliwatts of power.

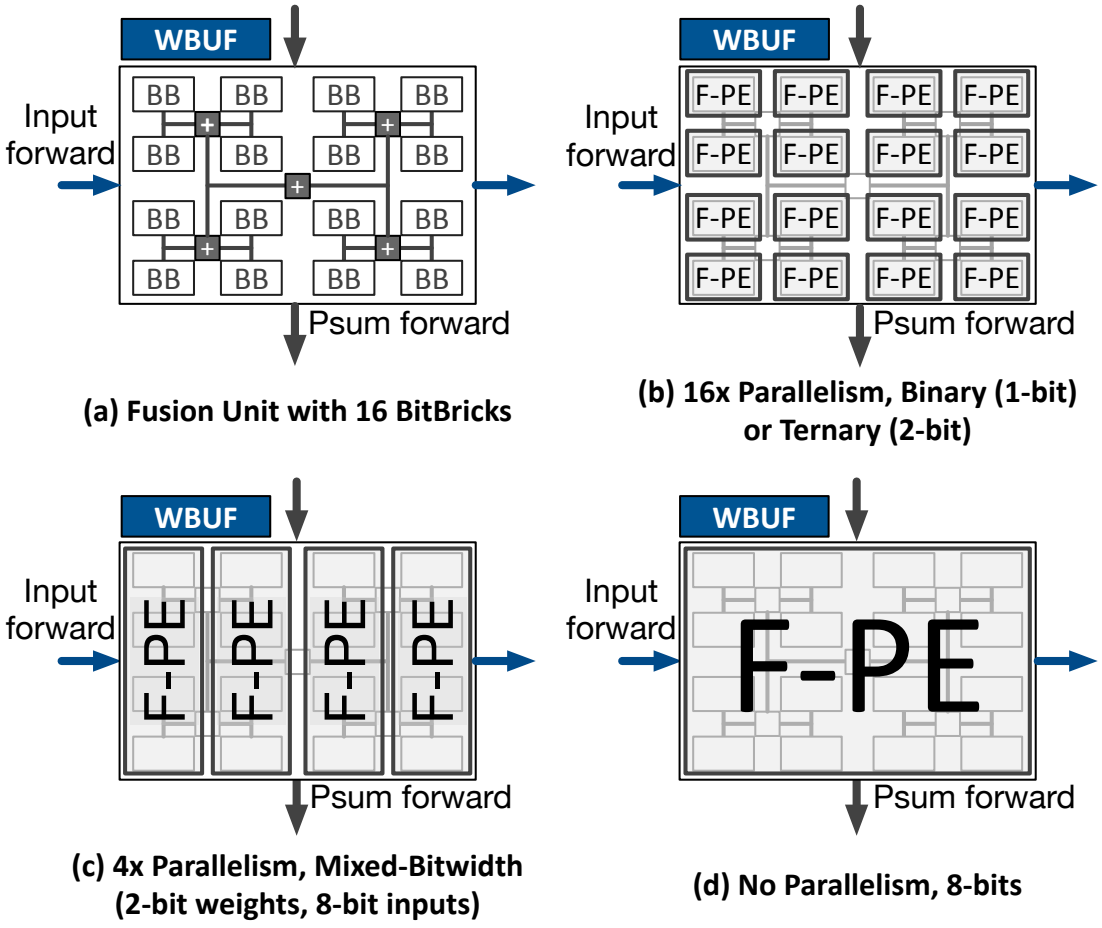


Figure 4.2: Dynamic composition of BitBricks (BBs) in a Fusion Unit to construct Fused Processing Engines (Fused-PE), shown as F-PE.

4.1.2 Bit Fusion Architecture

To minimize the computation and communication at the finest granularity, Bit Fusion dynamically matches the architecture of the accelerator to the bitwidth required for the DNN, which may vary layer by layer, without any loss in accuracy. As such, Bit Fusion is a collection of bit-level computational elements, called BitBricks, that dynamically compose to *logically* construct Fused Processing Engines (Fused-PE) that execute DNN operations with the required bitwidth. Specifically, Fused-PEs provide bit-level flexibility for multiply-adds, which are the dominant operations across all types of DNNs. Below, we discuss how BitBricks can be dynamically fused together to support a range of bitwidths, yet provide a significant increase in parallelism when operating at lower bitwidths.

Bit-Level Flexibility via Dynamic Fusion

As depicted in Figure 4.2, Bit Fusion arranges the BitBricks in a 2-dimensional *physical* grouping, called Fusion Unit. Each BitBrick in a Fusion Unit can perform individual binary (0, +1) and ternary (-1, 0, +1) multiply-add operations. As Figure 4.2 shows, the BitBricks *logically* fuse together at run-time to form Fused Processing Engines (Fused-PEs) that match the bitwidths required by the multiply-add operations of a DNN layer. The BitBricks in a Fusion Unit multiply an incoming variable-bitwidth input (input forward) to a variable-bitwidth weight (from WBUF) to generate the product. The Fusion Unit then adds the product to an incoming partial sum to generate an outgoing partial sum (`Psum forward` in Figure 4.2(a)).

Figures 4.2(b), 4.2(c), and 4.2(d) show three different ways of logically fusing BitBricks to form (b) 16 Fused-PEs that support ternary (binary); (c) four Fused-PEs that support mixed-bitwidths (2-bits for weights and 8-bits for inputs), (d) one Fused-PE that supports 8-bit operands, respectively. For binary or ternary operations (Figures 4.2(b)), each Fused-PE contains a single BitBrick, offering the highest parallelism. The Fusion Unit then adds the results from all Fused-PEs and the incoming partial sum to generate a single outgoing

partial sum. Figure 4.2(c) shows four BitBricks fused together in a column to form a Fused-PE that can multiply 2-bit weights with 8-bit inputs. The bitwidths of operands supported by a Fused-PE depend on the spatial arrangement of BitBricks fused together. Alternatively, by varying the spatial arrangement of the four fused BitBricks, the Fused-PE can support 8-bit/2-bit, 4-bit/4-bit, and 2-bit/8-bit configurations for inputs/weights. Finally, up to 16 BitBricks can fuse together to construct a single Fused-PE that can operate on 8-bit operands for the multiply-add operations (Figure 4.2(d)). The BitBricks fuse together in powers of 2. That is, a single Fusion Unit with 16 BitBricks can offer 1, 2, 4, 8, and 16 Fused-PEs with varying operand bitwidths. Dynamic composability of the Fusion Units at the bit level enables the architecture to expose the maximum possible level of parallelism with the finest granularity that matches the bitwidth of the DNN operands.

Accelerator Organization

Two insights guide the architecture design of Bit Fusion. First, DNNs offer high degrees of parallelism and benefit significantly from increasing the number of Fusion Units available within the accelerator's area budget. Therefore, it is essential to minimize the overhead of control in the accelerator by not only maximizing the number of Fusion Units but also minimizing the overhead of dynamically constructing Fused-PEs, thereby integrating the maximum number of BitBricks in the area budget. Second, on-chip SRAM and register-file accesses dominate the energy consumption when accelerating DNNs [157, 46, 47]. Therefore, it is essential to reduce the number of bits exchanged with on-chip and off-chip memory while maximizing data reuse.

Bit Fusion Systolic array. With these insights, we employ a 2-dimensional systolic array of Fusion Units as the architecture for Bit Fusion, as shown in Figure 4.3. The systolic organization reduces the overhead of control by sharing the control logic across the entire systolic array. More importantly, systolic execution alleviates the need for provisioning control for each Fused-PE as a dataflow architecture would have required. As such, the

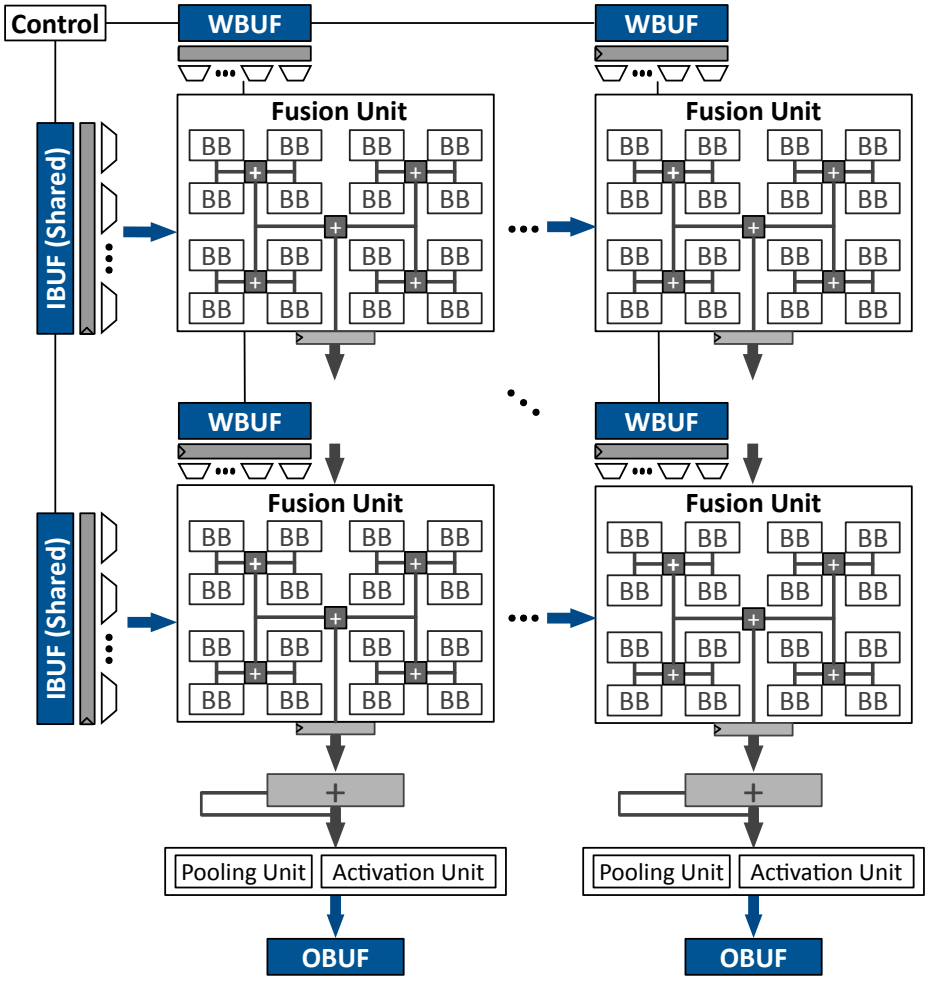


Figure 4.3: Bit Fusion systolic architecture comprising a collection of BitBricks (BBs) that can fuse to form Fused-PEs.

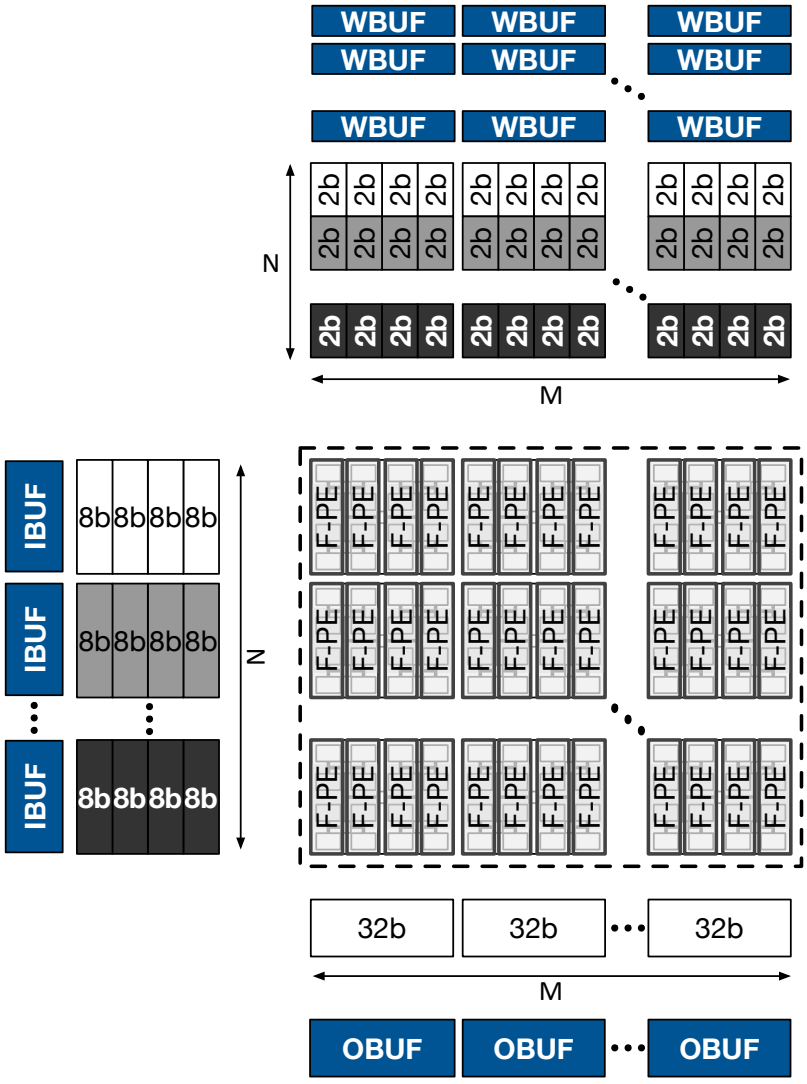


Figure 4.4: Bit-Flexible matrix-vector multiplication.

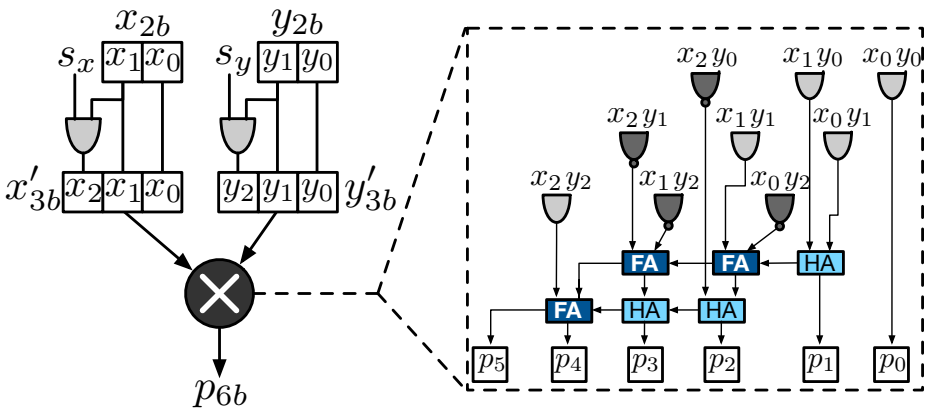


Figure 4.5: A single BitBrick. (HA: Half Adder, FA: Full Adder.)

systolic architectures fit the most number of BitBricks in a given area budget. Thus, the entire systolic array composed of Fused-PEs acts as a single compute unit that can execute, for example, a single matrix-vector multiplication operation with various bitwidths, which also sets the level of parallelism. In addition, the systolic organization of Fusion Units enforces sharing of input data across columns of the array and accumulates partial results across rows of the array to minimize access to on-chip memory. As depicted in Figure 4.3, the input buffers (`IBUFs`) only located at the borders and feed the rows simultaneously. Similarly, the output buffers (`OBUFs`) reside on the bottom and collect the flowing results, which is accumulated by each column's accumulator. As shown in Figure 4.3, each column harbors a pooling and an activation unit before its output buffer. Finally, the systolic organization also eliminates the need for local buffers for input, output, or partial results within Fusion Units. As such, each Fusion Unit is accompanied by only a weight buffer (`WBUF`). Using Fused-PEs as the building blocks, the performance of the systolic array maximally matches the bitwidths, with the highest performance at binary and ternary settings.

Memory organization. Depending on the number of Fused-PEs and their organization, the buffers must supply different number of operands with various bitwidths. As such, we augment the input and the weight buffers with a register that holds a row of data that is gradually fed to the Fused-PEs according to their bitwidth. As illustrated in Figure 4.3, a series of multiplexers after the register make this data infusion possible. The benefit of this

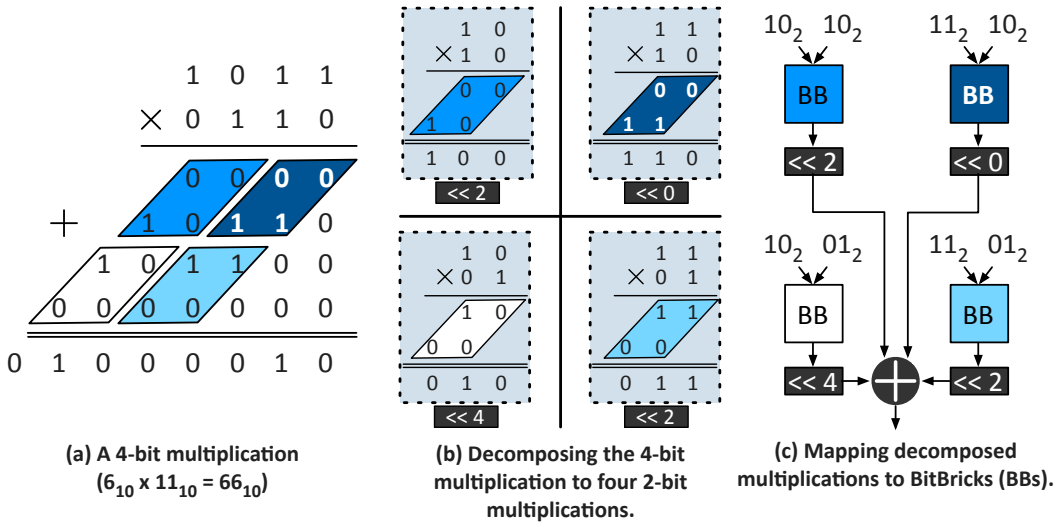


Figure 4.6: Using BitBricks to execute 4-bit multiplications.

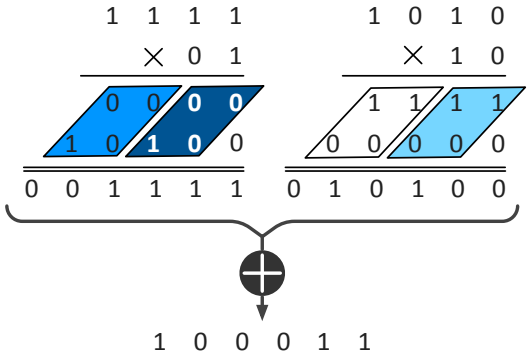


Figure 4.7: Two 4-bit $\times$ 2-bit multiplications decomposed to four 2-bit multiplications followed by the accumulation (summation) logic.

design is avoiding multiple accesses to the data array of the buffer which conserves energy. With this design, at each cycle, the systolic array consumes a vector of inputs and matrix of weights to produce a vector of outputs with the fewest accesses to the buffers and the minimal bitwidth possible.

Bit Fusion Execution Model

Figure 4.4 illustrates the Bit Fusion systolic execution in the mixed-bitwidth mode using when an input vector is multiplied to a weight matrix. The input vector has $4 \times N$ 8-bit elements that are being multiplied to a matrix with $4 \times N \times M$ 2-bit elements. As such, the 16-BitBricks in a Fusion Unit logically compose to form four 8×2 Fused-PEs. Both input and weight buffers provide 32 bits per access. The read values are split into 8-bit input

values and 2-bit weight values in the output register of each buffer using its accompanying multiplexers as mentioned before. The input values are shared across the Fusion Units of each row and weight values are specific to each Fused-PE. As such, all of the $4 \times N \times M$ Fused-PEs work in parallel while only a single 32-bit value is read from the input and weight buffers. Exploiting the lower bitwidth of weights, Bit Fusion increases the level of parallelism by $4\times$ while reducing the number of accesses to the weight buffer data arrays by the same factor of four. As discussed above, each Fusion Unit adds the results of its Fused-PEs with its incoming partials results and forwards the partial output to the Fusion Unit underneath it. As shown in Figure 4.4, we support 32-bit bitwidth for the partial and final results to avoid any inaccuracies.

4.1.3 Bit Fusion MicroArchitecture

Given the overall organization of Bit Fusion and its bit-flexible systolic execution model, this section delves into the details of BitBricks and Fusion Units. The key insight that enables bit-level dynamic composability in Bit Fusion is the mathematical property that a multiply operation between operands with power-of-2 bitwidths (4-bit, 8-bit, 16-bit, and so on) can be decomposed to 2-bit multiplications. The products from the decomposed multiplications can then be put together by shift-add operations to generate the results of the original multiplication. The bitwidths of the operands dictates the number of decomposed multiplications required and the shift amounts that are applied to the decomposed products before addition. Using this insight, we design BitBrick, the basic compute unit of the Bit Fusion architecture, to support multiply operations for the smallest bitwidth of 2-bits. The 2-bit operands for a BitBrick can be both signed or unsigned. Below, we describe the design of a single BitBrick.

BitBrick Microarchitecture

Figure 4.5 shows the microarchitecture of a single BitBrick. As shown, a BitBrick takes as input two 2-bit operands– x_{2b} and y_{2b} and two corresponding sign-bits–s_x and s_y. The sign-bits s_x and s_y define if the 2-bit operands are signed (between -2 to 1) or unsigned (between 0 to 3). According to the sign-bit, the BitBricks first extend the 2-bit operands x_{2b} or y_{2b} to respectively create 3-bit sign extended operands x'_{3b} or y'_{3b}. Finally, the BitBricks employ a 3-bit signed multiplier (shown with an encircled $\times$ in Figure 4.5) to generate a 6-bit product p_{6b}. Thus, a BitBrick supports both signed and unsigned numbers as its inputs. The following subsection discusses how Bit Fusion maps multiply-add operations with varying bitwidths to BitBricks.

Mapping Variable Bitwidth Operations to BitBricks

To explain how BitBricks compose to multiply operands with variable bitwidths, the discussion below uses a 4-bit multiplication as an example. As mentioned, a multiply operation with power-of-2 bitwidths can be decomposed to 2-bit multiplies that can execute using BitBricks. Figure 4.6(a) illustrates this mathematical property for a multiplication between 4-bit operands 1011_2 (11_{10}) and 0110_2 (6_{10}) to produce 01000010_2 (66_{10}). The 4-bit multiplication in Figure 4.6(a) decomposes to four 2-bit multiplications, shown in Figure 4.6(b). The decomposed multiplications execute using BitBricks to generate decomposed products, as shown in Figure 4.6(c). The decomposed products require shifting before being put together. For a 4-bit multiplication using BitBricks, the results from the decomposed 2-bit multiplications are left-shifted by 0, 2, 2, and 4, as shown in Figure 4.6(c).

Dynamic bitwidth flexibility. The bitwidths for the operands dictate how the results from the decomposed multiplications are left-shifted (multiplied with power of 2) before being added together. By adding flexibility in the shifting logic, the BitBricks can support 2-bit and even mixed-bitwidth (4-bit $\times$ 2-bit) multiplications. Figure 4.7 shows the summation of two 4-bit $\times$ 2-bit multiplications ($15_{10} \times 1_{10} + 10_{10} \times 2_{10} = 35_{10}$). The operation in

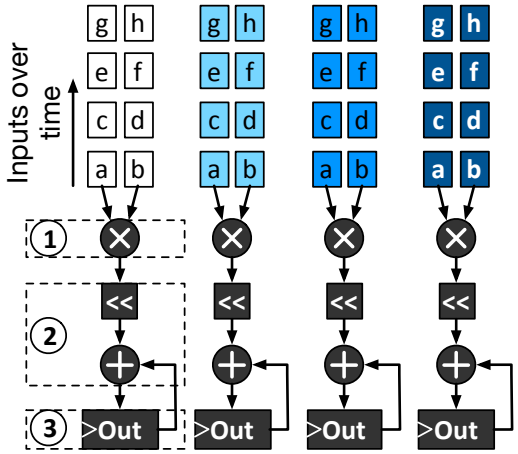


Figure 4.8: Temporal design. Operands $a - h$ are 2-bit.

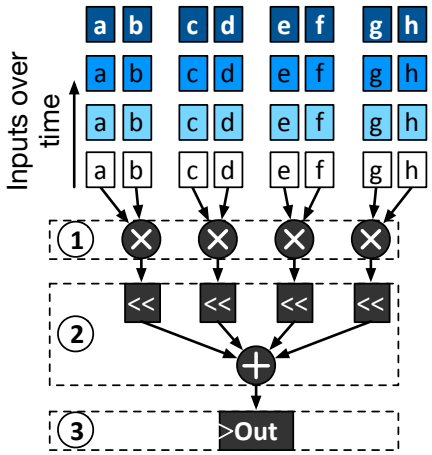


Figure 4.9: Spatial fusion. Operands $a - h$ are 2-bit.

Area (µm^2)	BitBricks	Shift-Add	Register	Total Area
Temporal	463	2989	1454	4905
Fusion Unit	369	934	91	1394
Area reduction over Temporal	1.3x	3.2x	16.0x	3.5x

Power (nW)	BitBricks	Shift-Add	Register	Total Power
Temporal	60	550	1103	1712
Fusion Unit	46	424	69	538
Power reduction over Temporal	1.3x	1.3x	16.0x	3.2x

Synthesized using a commercial 45 nm technology

Figure 4.10: Area and Power comparison of the Fusion Unit. Temporal design provided as reference.

Figure 4.7 breaks down to four 2-bit decomposed multiplications that map to four BitBricks. Both the single 4-bit $\times$ 4-bit operation in Figure 4.6(a) and the two 4-bit $\times$ 2-bit operations in Figure 4.7 require the same number of BitBricks. Therefore, the performance at 4-bit $\times$ 2-bit is twice that of 4-bit $\times$ 4-bit. The only difference between the operations in Figure 4.6(a) and Figure 4.7 is the shift amount required by the decomposed products. Similarly, when operating at 2-bit $\times$ 2-bit, each BitBrick can perform a single multiplication by setting the all the shift amounts to zero.

Supporting arbitrary bitwidths. The discussion so far shows how multiply operations between 4-bit and 2-bit operands map to BitBricks. The same mathematical property can

be recursively applied to support higher than 4-bit for the operands. Bit Fusion supports up to 16-bit operands by first recursively breaking down the 16-bit multiplication to 8-bit, 4-bit and then 2-bit multiplications which can execute using BitBricks. For a multiplication between $2n$-bit operands A_{2n} and B_{2n}, the recursion can be expressed mathematically as follows.

$$
\begin{aligned}
A_{2n} &= 2^n \times (A_{2n})_{hi} + 2^0 \times (A_{2n})_{lo} \\
B_{2n} &= 2^n \times (B_{2n})_{hi} + 2^0 \times (B_{2n})_{lo} \qquad (4.1) \\
A_{2n} \times B_{2n} &= 2^{2n} \times (A_{2n})_{hi} \times (B_{2n})_{hi} + 2^n \times (A_{2n})_{hi} \times (B_{2n})_{lo} \\
&\quad + 2^n \times (A_{2n})_{lo} \times (B_{2n})_{hi} + 2^0 \times (A_{2n})_{lo} \times (B_{2n})_{lo} \qquad (4.2)
\end{aligned}
$$

$(A_{2n})_{hi}$ and $(A_{2n})_{lo}$ refer to the n most significant and n least significant bits of A, respectively. By applying the above equation recursively, Bit Fusion supports up to 16-bit operands. When one of the operand's bitwidths is larger, we use the formulation below.

$$
A_{2n} \times B_n = 2^n \times (A_{2n})_{hi} \times B_n + 2^0 \times (A_{2n})_{lo} \times B_n \qquad (4.3)
$$

Each level of recursion, from 16-bits to 8-bits, 8-bits to 4-bits, and 4-bits to 2-bits, requires additional shift-add logic. The overhead from the shift-add logic represents the hardware cost of bit-level flexibility. The next subsection details the design of a Fusion Unit that uses BitBricks to execute multiply-adds with variable bitwidths, up to 16-bit.

Fusion Unit Micro-Architecture

To enable bit-level composability, Bit Fusion introduces *spatial fusion*, a paradigm that spatially combines the decomposed products generated by multiple BitBricks over a single cycle. Prior works [89, 168], on the other hand, devise a temporal design that use single-bit multiply-add units independently over the span of multiple cycles. The following elabo-

rates on these two approaches. To offer a fair comparison, we assume that even the temporal design uses 2-bit multipliers, a configuration that provides a better area, delay, and power as opposed to a fully bit-serial design.

Temporal design. Figure 4.8 shows a temporal design that can support variable bitwidths. The variable-bitwidth multiply operation for the temporal design consists of three steps: (1) 2-bit multiplication to generate a partial product, (2) shift operation to multiply with the appropriate power of 2, and (3) accumulation in a register. The temporal design requires 4 cycles to execute a 4-bit $\times$ 4-bit multiplication. The shift operation is simply a 4-input multiplexer (mux). Compared to a fixed 4-bit multiplier, the temporal design uses much smaller multiply units for 2-bit operands, which require significantly less area. However, the number of gates required for the shifter and the accumulator depend on the highest supported bitwidth (16-bit for Bit Fusion). For instance, to support up to 16-bits using a temporal design, the shifter and the accumulator use up around 90% of the area, which limits the benefits provided by this approach. Nevertheless, the temporal design reduces area consumption over a fixed-bitwidth multiplier for the highest required bitwidth.

Spatial fusion. In contrast, our spatial multiplier spatially combines (or fuses) the results from four BitBricks over a single cycle to execute either one 4-bit $\times$ 4-bit multiplication, two 4-bit $\times$ 2-bit multiplications, or four 2-bit $\times$ 2-bit multiplications. Figure 4.9 illustrates the design of a spatial multiplier that supports up to 4 bits for either of the two operands using BitBricks. Similar to the temporal design, the spatial multiplier requires three steps: (1) multiplication using BitBricks, (2) shift-add using the shift-add tree, and (3) accumulation of results in a register. The spatial multiplier improves upon the temporal design by using a shift-add tree and a single shared accumulator to reduce the number of gates required. Each level of the shift-add tree consists of three shift-units and a four-input adder that represent the multiplication with power of 2 in Equations (4.2) and (4.3). Compared to a 4-bit fixed bitwidth multiplier the spatial multiplier requires more area but delivers 4$\times$ higher performance for 2-bit operations. Overall, spatial fusion provides higher $\frac{performance}{area}$

compared to temporal design by packing more BitBricks in the same area.

Fusion Unit using spatio-temporal fusion. As discussed, a Fusion Unit can execute variable-bitwidth multiply-add operations and supports 2-bit to 16-bit operands. Using Equations (4.2) and (4.3) recursively, we can realize a Fusion Unit using either the temporal design, spatial fusion, or a combination of both. For a fixed area budget, using spatial fusion with 64 BitBrick would pack the highest number of BitBricks. At the same time, feeding the 64 BitBricks for spatial fusion would require 128-bit wide accesses to the SRAM buffers (IBUF and WBUF in Figure 4.3) per Fusion Unit. Increasing the width of SRAMs increases the area required by the IBUF and WBUF. Therefore, we make a tradeoff wherein we use spatial fusion to combine 16 BitBricks spatially to realize support up to 8-bit operands, and then combine it with temporal design to support up to 16-bit operands over four cycles. This hybrid approach balances both bit-level flexibility and the corresponding area overhead due to increased SRAM sizes. Figure 4.10 compares the area and the power requirements for a Fusion Unit with 16 BitBricks that uses the hybrid approach with a temporal design using 16 BitBricks. As shown, for 16 BitBricks, the hybrid Fusion Unit has 3.5× less area and 3.2× less power compared to temporal design with the same number of 2-bit multipliers.

Comparison to bit-serial temporal execution. Prior works in Stripes [89], UNPU [169], and Loom [168] devise bit-serial computation as a means to support flexible bitwidths for DNN operations. Of the three, Loom is a fully-temporal architecture, similar to the temporal design discussed above (Figure 4.8). Stripes and UNPU are hybrid designs that fix the bitwidth of one operand and support variable bitwidths for the other. We provide a head-to-head comparison to Stripes in Section 4.3 and provide a qualitative comparison to Loom below. As the results from Figure 4.10 indicate, for the same throughput, a fully-temporal design, such as the one used in Loom, would consume significantly larger area and power compared to our spatially composable Fusion Unit. Furthermore, a fully-temporal design iterates in the form of a nested loop over the bits the two operands; hence, requiring more number of accesses to the SRAM.

Table 4.1: Bit Fusion Instruction Set.

OpCode	Operand Specification		Loop Identifier	Immediate
5-bits	6-bits		5-bits	16-bits
setup	op0.bitwidth	op1.bitwidth	X	X
ld-mem	scratchpad-type	mem.bitwidth	loop-id	num-words
st-mem				
rd-buf		X		X
wr-buf				
gen-addr		ld/st		stride
compute	fn			X
loop	X	loop-level		num-iterations
block-end	Address of next instruction			

The next section discusses the Bit Fusion-ISA, that exposes the bit-level flexibility of Bit Fusion to software.

4.1.4 Bit Fusion Instruction Set Architecture - Fusion-ISA

To leverage the unique bit-level flexibility of Bit Fusion, we need to design a new hardware-software interface that exposes those capabilities in an abstract manner. Furthermore, the abstraction must be flexible to enable a wide range of DNN models so as to exploit bit-level fusion. The following lists the requirements for an ISA that provides this abstraction and enables efficient use of Bit Fusion for various categories of DNNs.

1. **Amortize the cost of bit-level fusion by grouping operations.** The operations in a DNN are organized into groups, called layers, wherein the same mathematical operation repeats a large number of times (often hundreds of thousands). To avoid the overhead of *fine-grained* control over the operations at such a scale, the abstraction needs to amortize the cost of bit-level fusion across blocks of instruction that implement the layers.
2. **Enable a flexible data-path for Bit Fusion.** Both the number of words and the bitwidth of each word that feeds the Fused-PEs varies depending on how the BitBricks are composed as discussed in Section 4.1.2. Thus, the semantics of instructions for data accesses must vary according to the fusion configuration to enable a flexible data-path.

3. **Provide a concise expression for a wide range of DNN layers**. As research in DNNs is still volatile, it is necessary to devise an ISA that is general enough to express a wide range of DNN operations/layers. Yet, minimizes the von Neumann overhead of instruction handling and require a small footprint.

Fusion-ISA for Bit-Flexible Acceleration

Table 4.1 summarizes the Bit Fusion-ISA that aims to satisfy these requirements. The rest of this section discusses the instruction formats and provides the insight that drives them.

Block-structured ISA for DNN layers. To leverage the commonalities in the operations of a layer, the Bit Fusion ISA is *block structured*. As such, the fusion configuration of the BitBricks is fixed across each block of instructions that implement a specific layer. In this work, we did not explore within layer bitwidth variations. Nevertheless, the Bit Fusion ISA and this incarnation of its microarchitecture can readily support it by using multiple instruction blocks for an individual layer. The `setup` instruction marks the beginning of an instruction block and configures the Fusion Units and its data delivery logic to the specified bitwidth for the operands. This instruction effectively defines the *logical* fusion of the BitBricks into Fused-PEs for all the instructions in the block. The `block-end` instruction signifies the end of a block and provides the address to the next instruction in the `next-inst` field.

Concise expression of DNN layers. DNNs consist of a large number of simple operations like multiply-accumulate and max, repeated over a large number of neurons (over 2600 million multiply-adds in AlexNet. See Table 4.2). Thus, the von Neumann overhead of instruction fetch and decode can limit performance due to the large number of operations required by a DNN. To minimize the number of instruction fetches/decodes required, we leverage the following insight. Each layer in a DNN is a series of simple mathematical operations over hyper-dimensional arrays. How the operations walk through the array elements and the type of mathematical operation (multiply-add/max) uniquely defines a

layer. As such, the ISA provides `loop` instructions that enable a concise way of defining the walks and operations in a DNN layer. Each `loop` instruction has a unique ID in the block. As shown in Table 4.1, the `num-iterations` field in the `loop` instruction defines iteration count. The `compute` instruction specifies the type of operation, while the `gen-addr` instruction dictates how to walk through the elements of the input/output hyper-dimensional arrays. The `stride` field in the `gen-addr` instruction specifies how to walk through the array elements in the `loop`, which is identified by the `loop-id` field. The words after the `setup` instruction define the memory base address for the data that fills the three buffers of input, output, and weights. The `gen-addr` instruction generates the addresses that walk through the memory data and fill the buffers.

$$Address = base + \sum_{id}(loop\_iterator[id] \times stride[id]) \tag{4.4}$$

In Equation (4.4), id is the `loop-id` field of all the `gen-addr` instruction in the block and the $loop\_iterator$ is the current iteration of the corresponding loops and their $stride$s. The fundamental assumption is that multiple `gen-addr` instructions repeated by corresponding `loop` instructions define the complex multi-dimensional walks that expresses various kinds of DNN layers from LSTM to CNN. In the evaluated benchmarks, blocks with 30-86 instructions are enough to cover LSTM, CNN, pooling, and fully connected. These blocks use a combination of `loop`, `compute`, and `gen-addr` instructions to define these DNN layers nested loops. These statistics show that our ISA can concisely express various DNN layers while providing bit-level fusion capabilities. Note that these instructions are fetched and decoded once at the beginning of an instruction block, amortizing the von Neumann overhead over the entire execution of the block.

Managing memory accesses for Fused-PEs. The `ld-mem`/`st-mem` instructions exchange data between the on-chip buffers (`IBUF`, `OBUF`, and `WBUF`) in Figure 4.3–and the off-chip memory. Similarly, the `rd-buf`/`wr-buf` instructions read/write data from the

on-chip buffers specified by the `scratchpad-type` field as shown in Table 4.1. In these four instructions, the size of the operands, which are variable-bitwidth arrays, depends on the number of array elements and their bitwidths. These parameters, which control the logic that feeds the Fused-PEs, are dependent on the bit-level fusion configuration (number of Fused-PEs in each Fusion Unit) and the type of data (input/weights). To capture this variation in the size of data, the semantics of `rd-buf`/`wr-buf` and `ld-mem`/`st-mem` instructions for accessing on-chip and off-chip memory vary according to the fusion configuration of their instruction block, set apriori. In particular, the sizes of memory accesses by `ld-mem`/`st-mem` instructions depend on both its `num-words` field and the fusion configuration defined by the corresponding `setup` instruction.

Decoupling on-chip and off-chip memory accesses. The data required by DNNs, and subsequently, the number of memory accesses are large. Hence, the latency due to off-chip memory accesses can be a performance bottleneck. To hide the latency of off-chip accesses, the ISA decouples the on-chip memory accesses with off-chip. Furthermore, decoupling the two types of memory accesses allows the accelerator to reuse on-chip data using simple scratchpad buffers, instead of hardware-managed caches.

Code Optimizations

As discussed in Section 4.1.4, the Fusion-ISA uses simple instructions combined with explicit loop instructions to express neural networks. The use of simpler instructions makes the ISA flexible to express a large range of DNNs. Nonetheless, the flexibility in the ISA enables incorporating layer-specific optimizations to improve the performance and energy gains. For brevity, we use an example fully-connected layer to discuss the code optimizations. Figure 4.11 shows the matrix-matrix multiplication associated with this example. We perform the following three optimizations as depicted in Figure 4.12.

Loop ordering. Loop-ordering optimizes the order of the outer loops and memory instructions to further reduce off-chip accesses. Recall that Bit Fusion-ISA uses loop indices

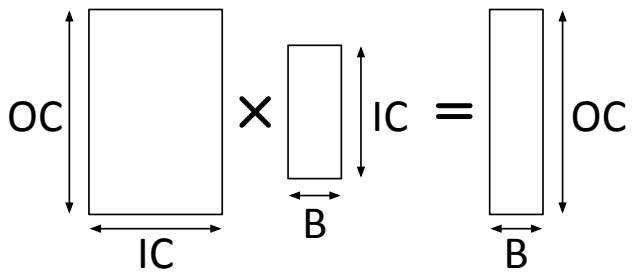


Figure 4.11: A single Fully-Connected Layer. The $\times$ represents matrix multiplication.

to generate memory addresses (Section 4.1.4). When the address for a memory instruction does not depend on the index of the previous loop instruction, their order can be exchanged. The optimized code in Figure 4.12(b) uses `Output-Stationary` for executing the fully-connected layer, to reduce read/write accesses to the output buffer. Changing the order allows Bit Fusion to switch between `Input-Stationary`, `Output-Stationary`, and `Weight-Stationary` to minimize off-chip and on-chip accesses.

Loop tiling. Loop-tiling partitions a loop instruction in the Bit Fusion-ISA into smaller *tiles* such that the data required by a loop operation fits inside the on-chip scratchpads. The smaller tiles are accessed using a single LD/ST instruction and are reused in the inner-loop to reduce off-chip accesses. Compared to the original code in Figure 4.12(a), the tiled version in Figure 4.12(b) reduces off-chip accesses for output buffer by a factor of `IC`$\times$, and on-chip accesses for output buffer by a factor of $tile_{ic}$. Note that `IC` is a dimension in the matrix multiplication operation as depicted in Figure 4.11. Convolution layers typically require six loop instructions, which increases to 12 after tiling optimizations. The overhead of increasing the number of instructions on performance is negligible since the cost of fetch and decode is amortized throughout the execution of the layer.

Layer fusion. As discussed, the Bit Fusion architecture consists of a 2-D systolic array of multipliers, along with a 1-D array of pooling/activation units. When two or more consecutive layers use mutually exclusive on-chip resources, the instructions for the two layers are combined such that the data produced by the first layer is directly fed into the subsequent layer, avoiding costly off-chip accesses. For example, the fully-connected layer in

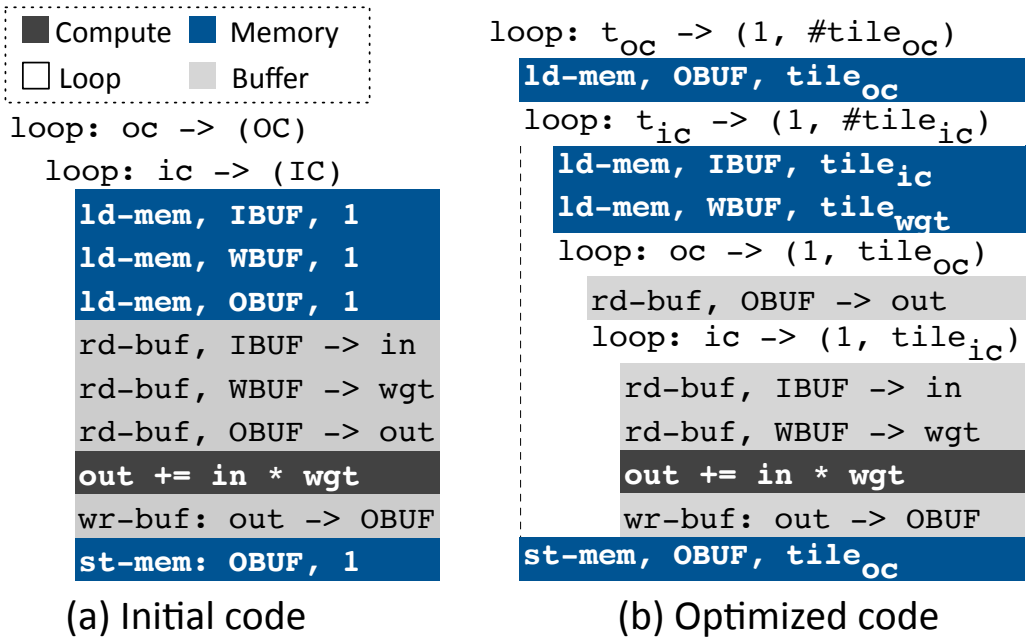


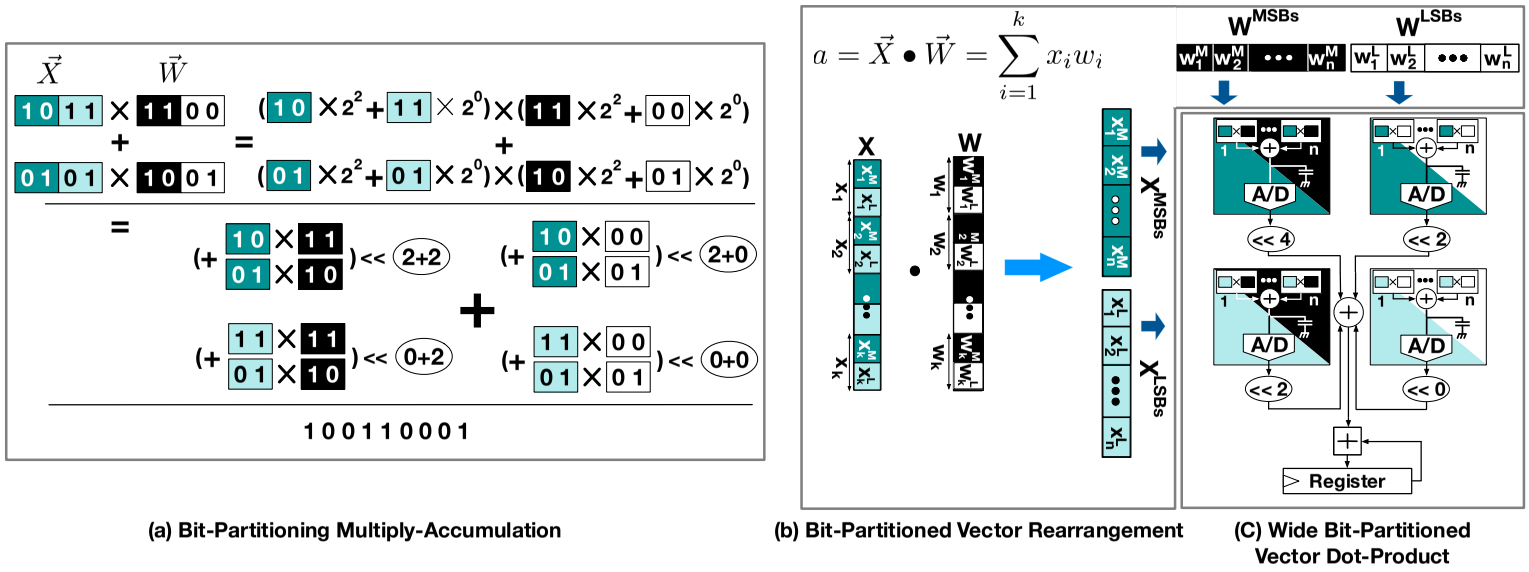
Figure 4.12: (a) Code for the Fully-Connected Layer. (b) Optimized code using loop tiling and ordering. `setup` and `gen-addr` instructions omitted for clarity.

Figure 4.11 uses the 2-D systolic array. If the next layer is activation, then we can fuse the layers and create one block of instruction for computing both the layers.

4.2 Enabling Mixed-Signal Acceleration through Bit-Partitioned Arithmetic

The discussion so far in this chapter uses the mathematical properties of the convolutions and fully-connected layers in DNNs to develop a bit-level composable architecture using low-bitwidth digital compute units. The key insight is that the multiply-adds for convolutions and fully-connected layers with varying bitwidths for its operands can be decomposed into low-bitwidth multiply-adds. This section takes an alternative approach and uses the mathematical properties of maps of vector-dot-product, the prevalent operation in DNNs, to construct low bitwidth analog-domain compute units, enabling mixed-signal acceleration. The following section discusses the insights and mathematical formulation in detail.

Bit-Level partitioning and interleaving of MACCs. Figure 4.13(a) delves into the bit-level operations of dot-product on vectors with 2-elements containing 4-bit values. As illustrated with different colors, each 4-bit element can be written in the form of sum of 2-bit



(a) Bit-Partitioning Multiply-Accumulation

(b) Bit-Partitioned Vector Rearrangement

(C) Wide Bit-Partitioned Vector Dot-Product

Figure 4.13: Wide, interleaved, and bit-partitioned mathematical formulation.

partitions multiplied by powers of 2 (shift). As discussed, vector dot-product is also a sum of multiplications. Therefore, by utilizing the distributive property of addition and multiplication, we can rewrite the vector-dot product in terms of the bit partitions. However, we also leverage the associativity of the addition and multiplication to group the bit-partitions in the same positions together. For instance, in Figure 4.13, the black partitions that represent the Most Significant Bits (MSBs) of the $\vec{W}$ vector are multiplied in parallel to the teal[1] partitions, representing the MSBs of the $\vec{X}$. Because of the distributivity of multiplication, the shift amount of (2+2) can be postponed after the bit-partitions are multiply-accumulated. The different colors of the boxes in Figure 4.13 illustrates the interleaved grouping of the bit-partitions. Each group is a set of spatially parallel bit-partitioned MACC operations that are drawn from different elements of the two vectors. The low-bitwidth nature of these operations enables execution in the analog domain without the need for A/D conversion for each individual bit-partitioned operation. As such, our proposed reformulation amortizes the cost of A/D conversion across the bit-partitions of different elements of the vectors as elaborated below.

Wide, interleaved, and bit-partitioned vector dot-product. Figure 4.13(b) illustrates the proposed vector dot-product operation with 4-bit elements that are bit partitioned to 2-bit sub-elements. For instance, as illustrated, the elements of vector X, denoted as x_i, are first bit partitioned to x_i^L and x_i^M. The former represents the two Least Significant Bits (LSBs) and the latter represents the Most Significant Bits (MSBs). Similarly, the elements of vector W are also bit partitioned to the w_i^L and w_i^M sub-elements. Then, each vector (e.g., W) is rearranged into two bit-partitioned sub-vectors, W^{LSBs} and W^{MSBs}. In the current implementations of BIHIWE architecture, the size of bit-partition is fixed across the entire architecture. Therefore, the rearrangement is just rewiring the bits to the compute units that imposes modestly minimal overhead (less than 1%). Figure 4.13 is merely an illustration and there is no need for extra storage or movement of elements. As

[1]Color teal in Figure 4.13 is the darkest gray in black and white prints.

depicted with color coding, after the rewiring, W^{LSBs} represents all the least significant bit-partitions from different elements of vector W, while the MSBs are rewired in W^{MSBs}. The same rewiring is repeated for the vector X. This rearrangement, puts all the bit-partitions from all the elements of the vectors with the same significance in one group, denoted as W^{LSBs}, W^{MSBs}, X^{LSBs}, X^{MSBs}. Therefore, when a pair of the groups (e.g., X^{MSBs} and W^{MSBs} in Figure 4.13(c)) are multiplied to generate the partial products, (1) the shift amount ("$\ll 4$" in this case) is the same for all the bit-partitions and (2) the shift can be done after partial products from different sub-elements are accumulated together.

As shown in Figure 4.13(c), the low-bitwidth elements are multiplied together and accumulated in the analog domain. Accumulation in the digital domain would require an adder tree which is costly compared to the analog accumulation that merely requires connectivity between the multiplier outputs. It is only after several analog multiply-accumulations that the results are converted back to digital for shift and aggregation with partial products from the other groups. The size of the vectors usually exceeds the number of parallel low-bitwidth MACCs, in which case the results need to be accumulated over multiple iterations. The accumulations are performed in two steps. The first step accumulates the results in the analog domain through charge accumulation in capacitors before A/D convertors (see Figure 4.13(c)). In the second step, these converted accumulations will be added up in the digital domain using a register. For this pattern of computation, we are effectively utilizing the *distributive and associative property* of multiplication and addition for dot-product but at the *bit granularity*. This rearrangement and spatially parallel (i.e., wide) bit-partitioned computation is in contrast with temporally bit-serial digital [89, 158, 169, 168] and analog [163] DNN accelerators.

4.2.1 Evaluation

This section provides a thorough evaluation of the benefits from bit-level composability offered by the Bit Fusion architecture using purely-digital circuits.

Methodology

Benchmarks. Table 4.2 shows the list of 8 CNN and RNN benchmarks from diverse domains including image classification, object and optical character recognition, and language modeling. The selected DNN benchmarks use a diverse size of input data, which allows us to evaluate the effect of input data size on the Bit Fusion architecture. AlexNet [170, 156], SVHN [171, 155], CIFAR10 [57, 155], LeNet-5 [172, 154], VGG-7 [60, 154], ResNet-18 [173, 156] are popular and widely-used CNN models. Among them, AlexNet and ResNet-18 benchmarks are image classification applications that have different network topologies that use the ImageNet dataset. The SVHN and LeNet-5 benchmarks are optical character recognition applications that recognize the house numbers from the house view photos and handwritten/machine-printed characters, respectively. CIFAR10 and VGG-7 are object recognition applications based on the CIFAR-10 and ImageNet dataset, respectively. The RNN [155] and LSTM [174, 155] are recurrent networks that perform language modeling on the Penn TreeBank dataset [175]. In Table 4.2, the "Multiply-Add Operations" column shows the required number of Multiply-Add operations for each model and the "Model Weights" column shows the size of model parameter. Note that the multiply-add operations and model weights have variable bitwidths as presented in Figure 4.1.

Reduced bitwidth DNN models. Bit Fusion aims to accelerate the inference of a wide range of DNN models with varying bitwidth requirements, with *no loss in classification accuracy*. The benchmarks, listed in Table 4.2, employ the model topologies proposed in prior work [152, 155, 154, 156] that train low bitwidth DNNs and achieve the same accuracy as the 32-bit floating-point models. We did not engineer these quantized DNNs and merely took them from the existing deep learning literature [152, 155, 154, 156]. Benchmarks Cifar-10, SVHN, LSTM, and RNN use the quantized models presented in [155]. Benchmarks LeNet-5 and VGG-7 use ternary (+1,0,-1) networks [154]. AlexNet and ResNet-18 use the 4-bit $2\times$ wide models presented in [156] that double the number of channels for convolution

Table 4.2: Evaluated CNN/RNN benchmarks.

DNN	Type	Domain	Dataset	Multiply-Add Operations	Model Weights
AlexNet	CNN	Image Classification	ImageNet	2,678 Mops	116.3 Mbytes
Cifar-10	CNN	Object Recognition	CIFAR-10	617 Mops	3.3 MBytes
LSTM	RNN	Language Modeling	Penn TreeBank	13 Mops	6.2 MBytes
LeNet-5	CNN	Optical Character Recognition	MNIST	16 Mops	0.5 MBytes
ResNet-18	CNN	Image Classification	ImageNet	4,269 Mops	13.0 Mbytes
RNN	RNN	Language Modeling	Penn TreeBank	17 Mops	8.0 MBytes
SVHN	CNN	Optical Character Recognition	SVHN	158 Mops	0.8 MBytes
VGG-7	CNN	Object Recognition	CIFAR-10	317 Mops	2.7 MBytes

Table 4.3: Evaluated ASIC and GPU platforms. *Stripes entries per-tile.

	ASIC			GPU	
Chip	**Eyeriss**	**Stripes***	**Chip**	**Titan X**	**Tegra X2**
Cores (1.1 mm^2)	168 PEs	4096 SIPs	**Cores**	3,584	256
On-chip Memory	181.5 KB	2 MB eDRAM 16 KB SRAM	**Memory**	12 GB	8 GB
Chip Area (mm^2)	5.87	3.62	**ChipArea (mm^2)**	471	-
			TDP	250 W	7.5 W
Frequency	500 MHz	980 MHz	**Frequency**	1,531 MHz	875 MHz
Technology	45 nm	45 nm	**Technology**	16 nm	16 nm

and fully-connected layers. We use the regular AlexNet and ResNet-18 models for Eyeriss and the GPU baselines, and use their $2\times$ wide quantized models for Bit Fusion and Stripes.

Accelerator development and synthesis. We use RTL-Verilog to implement the configuration of the Bit Fusion architecture and verify the design through extensive RTL-simulations. We synthesize Bit Fusion at 45nm technology node using Synopsys Design Compiler (L-2016.03-SP5) and a commercial standard-cell library. Design Compiler provides the chip area, achievable frequency, and dynamic/static power, which we use to estimate the performance and energy-efficiency of the Bit Fusion accelerator.

Simulation infrastructure for Bit Fusion. We compile each DNN benchmark to the instructions of the Fusion-ISA (Section 4.1.4). We develop a cycle-accurate simulator that takes the Fusion-ISA instructions for the given DNN and simulates the execution to calculate the cycle counts as well as the number of accesses to on-chip buffers (IBUF, OBUF, and WBUF in Figure 4.3) and off-chip memory. We verify the cycle counts of the simulator

against our Verilog implementation of the Bit Fusion architecture. Using the frequency defined in Table 4.3 and the cycle counts, the simulator measures the execution time of the Bit Fusion architecture. To evaluate the energy efficiency, we model the energy consumption for on-chip buffers for the Bit Fusion accelerator using the results from CACTI-P [176].

Comparison with Eyeriss. To measure the performance and energy dissipation of our comparison point, Eyeriss, we use their open-source simulation infrastructure [157]. The resulting area and energy metrics are shown in Table 4.3. As mentioned, we use the same area budgets as Eyeriss, which is 1.1 mm^2 for compute units and 5.87 mm^2 for chip to synthesize Bit Fusion, shown in Table 4.3. We use a total 112 KB SRAM for on-chip buffers (IBUF, OBUF, and WBUF in Figure 4.3). Eyeriss operates on the 16-bit operands and Bit Fusion supports flexible bitwidths from 2, 4, 8, to 16 bits.

Comparison with Stripes. The authors of Stripes graciously shared their simulator [89]. Their power estimation tools were in 65 nm node, which we scaled to 45 nm. Stripes operates on 16-bit inputs and variable-bitwidth weights (1 through 16), using Serial Inner-Product units (SIPs). Stripes is organized into 16 tiles each of which has 4096 SIPs. For a fair comparison, we replace the 4096 SIPs in each tile of Stripes with our proposed Bit Fusion systolic array with 512 Fusion Units, each with 16 BitBricks to match the same budget of 1.1mm^2 for compute, which is the area after scaling to 45 nm and use the same total on-chip memory.

Comparison with GPUs. We use two GPUs (Titan Xp and Tegra X2) based on Nvidia's Pascal architecture to compare with Bit Fusion. Table 4.3 shows the details of the two GPUs. We use Nvidia's custom TensorRT 4.0 [177] library compiled with the latest CUDA 9.0 and cuDNN 7.1 which support 8-bit quantized calculations, the smallest possible in the architecture. Across GPU platforms, we use 1,000 warm-up batches, followed by 10,000 batches to measure performance and use the average. For a head-to-head comparison, we conservatively scale Bit Fusion to 16 nm technology node assuming a $0.86\times$ voltage scaling and $0.42\times$ capacitance scaling according to the methodology presented in [33]. However,

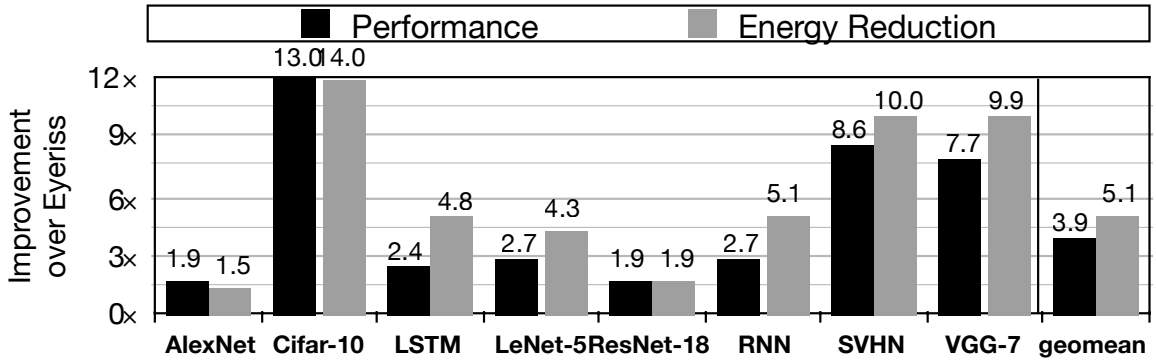


Figure 4.14: Bit Fusion performance and energy improvements over Eyeriss.

we assume the same frequency of 500 MHz as Eyeriss and do not increase the Bit Fusion frequency. The scaled Bit Fusion architecture has 4096 Fusion Units with 896 KB SRAM and has a total chip area of 5.93 mm^2 and consumes 895 milliwatts of power. As a point of reference, Titan Xp in the same 16 nm node, has a chip area of 471 mm^2 and has a TDP of 250 Watts, as summarized in Table 4.3.

4.3 Experimental Results

Comparison to Eyeriss

Performance and energy improvement. To evaluate the performance and efficiency benefits from the Bit Fusion architecture, we compare with a state-of-the-art accelerator Eyeriss [46] that proposes an optimized dataflow architecture for DNNs. We match the same area budget of 1.1mm^2 for computational logic across both architectures: systolic array in Bit Fusion and PEs in Eyeriss, and match the total SRAM capacity. We scale the area and energy consumption of the PEs, register-files, on-chip network, and DRAM in Eyeriss to 45nm technology according to the methodology proposed in [157]. For a fair comparison between the two architectures, we use the same frequency of 500MHz reported in the paper [157] for both Eyeriss and Bit Fusion. Figure 4.14 presents the performance and energy benefits of Bit Fusion in comparison with Eyeriss. On average, Bit Fusion delivers 3.9× speedup since the Bit Fusion architecture can perform more DNN operations with lower

bitwidth in a given area compared to Eyeriss. Depending on the types of DNN operations and the required bitwidths, the benchmarks see different performance gains. The CNN benchmarks (AlexNet, SVHN, Cifar-10, LeNet-5, VGG-7, and ResNet-18) see higher performance gains than the recurrent networks (RNN and LSTM) since the convolution operations are more amenable for data reuse in systolic architecture of Bit Fusion. Cifar-10 sees the highest benefits of 13× speedup since most of its operations can be computed with the smallest bitwidth (1-bit input and 1-bit weight) and its operations provide a large degree of parallelism that can exploit the increased number of Fused-PEs. In contrast, ResNet-18 and AlexNet achieve the lowest speedup of 1.9×, because these two benchmarks use twice the number of channels ($2\times$ wide) for convolution and fully-connected layers [156] for quantized execution on Bit Fusion. We use the original AlexNet and ResNet-18 models on Eyeriss, which effectively requires $4\times$ less multiply-add operations. Overall, using variable bitwidth improves performance and energy efficiency, since it increases compute capacity and reduces active hardware components. Figure 4.14 also shows the energy reduction. The average improvement is 5.1×, with the largest of 14× from Cifar-10 and the smallest of 1.5× from AlexNet. The significant energy reduction attributes to both Fusion Unit organizations and memory access reductions, which we discuss below in more detail.

Energy breakdown. To understand the sources of the energy reduction, we break down the energy consumptions for each hardware component (compute units, on-chip SRAM buffers, register file, and off-chip DRAM memory). Figure 4.15 shows the per-component energy dissipation for Bit Fusion and Eyeriss. This figure should be considered with the energy reduction results from Figure 4.14. Both accelerators consume more than 80% of energy for on-chip and off-chip memory accesses. The bit-level flexibility for memory accesses in Bit Fusion significantly reduces energy consumption for both on-chip buffers (IBUF, OBUF, and WBUF in Figure 4.3) and off-chip DRAM. Furthermore, with bit-level flexibility, our buffers can hold more data at lower-bitwidths, effectively giving Bit Fusion more on-chip storage capacity, which leads to fewer off-chip memory accesses. Eyeriss

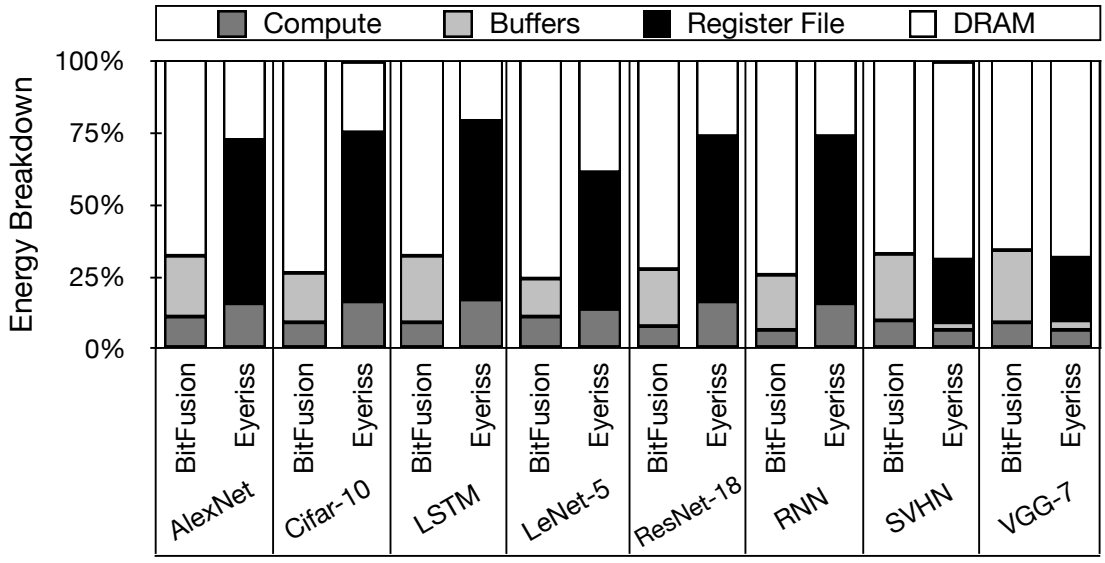


Figure 4.15: Energy breakdown of Bit Fusion and Eyeriss.

employs local register files within each PE, which constitutes a significant portion of the energy consumption. Bit Fusion's systolic architecture avoids the need for register files and enforces explicit data sharing for inputs and partial results, as shown in Figure 4.3. Therefore, Bit Fusion saves on Register File energy, but requires more SRAM accesses. The combined effect of bit-level flexibility and the systolic organization of BitBricks in the Bit Fusion architecture provides an average energy savings of 5.1×. Off-chip DRAM accesses, however, are still a significant portion of Bit Fusion's energy consumption and its share grows due to the significant reduction of compute and on-chip storage energy.

Sensitivity Study

Sensitivity to memory bandwidth. Depending on the DNN topology, the impact of off-chip bandwidth on performance varies. To understand the correlation between bandwidth and performance, we perform a sensitivity study for bandwidth. Figure 4.16 shows the performance improvements with Bit Fusion as we change the bandwidth from 0.25× to 4× of the default value. The baseline in this study the Bit Fusion with the default bandwidth of 128 bits per cycle. On average, when we scale the bandwidth up to 4×, Bit Fusion

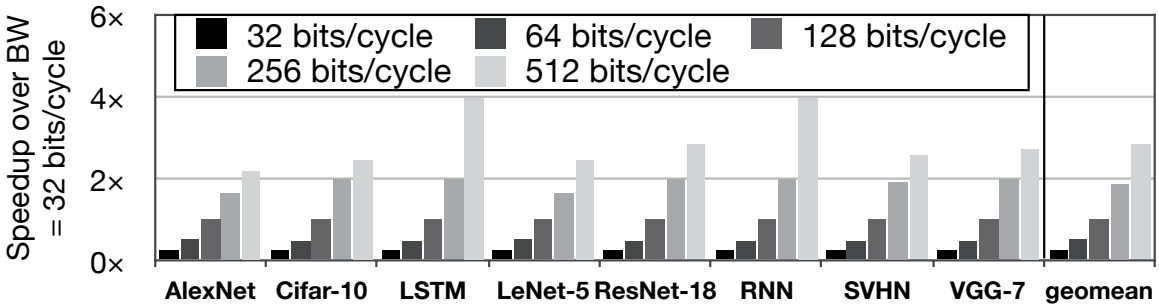


Figure 4.16: Bit Fusion performance as the bandwidth changes.

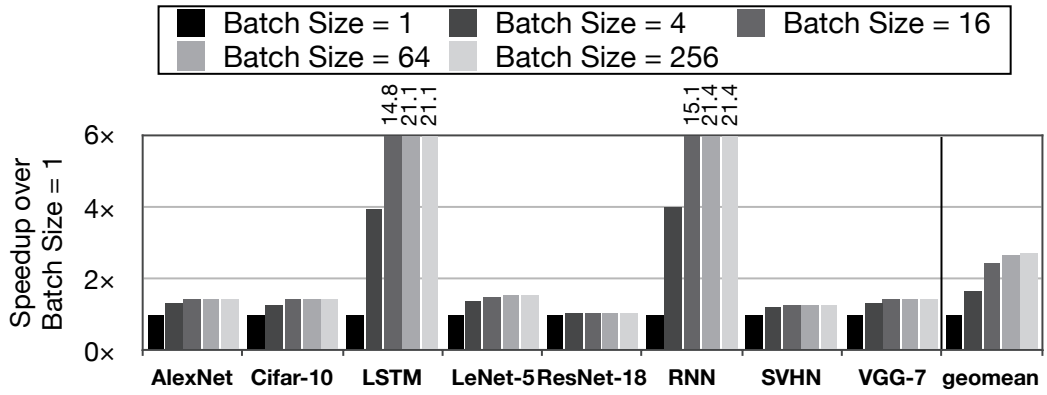


Figure 4.17: Bit Fusion performance as the batch size increases.

provides 1.6× speedup compared to the default setting, while with 0.25× bandwidth, the performance degrades 60%. Since CNN benchmarks see more opportunities for data reuse, they have less sensitivity to the bandwidth compared to the RNN benchmarks. The two RNN benchmarks, LSTM and RNN, provide almost linearly-scaling speedup as they are bottlenecked by the bandwidth.

Sensitivity to batch size. Batching amortizes the cost of weight reads by sharing weights across a batch of inputs. Figure 4.17 shows how performance changes as we increase batch size from 1 through 256 with the batch size 1 as the baseline (no batching). Our default batch size is 16. On average, Bit Fusion with the batch size of 256 engenders 2.7× speedup with the highest speedup of 21.4× from RNN. Since batching is effective when the bandwidth is limited and the performance is bandwidth-bound, the trends are similar to the bandwidth sensitivity results presented in Figure 4.16. However, there is a marginal gain across all the benchmarks when the batch is increased from 64 to 256, since beyond a batch

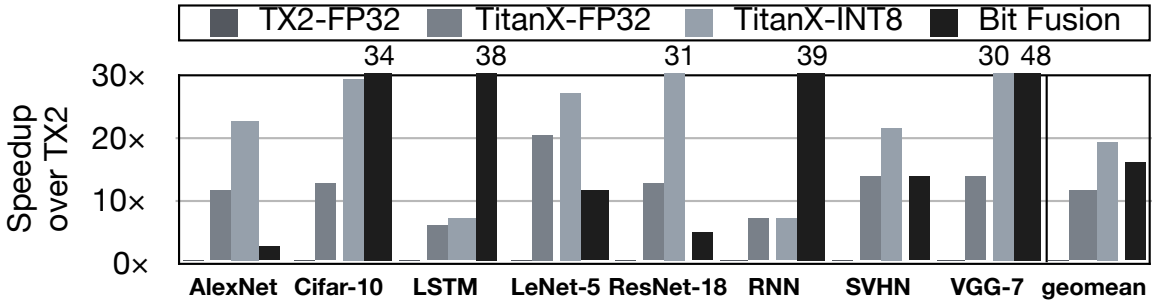


Figure 4.18: Performance comparison to GPUs.

size of 64, the bandwidth is sufficient to keep all the Fusion Units occupied.

Comparison to GPUs

Performance comparison to GPUs. GPUs are the most widely-used general-purpose processors for DNNs. We compare the performance of Bit Fusion accelerators with two GPUs: (1) Tegra X2 (TX2), and (2) Titan X based on the Pascal architecture (Titan Xp), details of which are presented in Table 4.3. As mentioned in the methodology Section 4.2.1, we scale Bit Fusion to match the 16 nm technology node of the GPUs, and use a total of 4096 Fusion Units. Figure 4.18 shows the speedup of TitanX and Bit Fusion using the TX2 as the baseline. TX2 does not support 8-bit mode natively. Due to this lack of support, empirical results show slow down when the 8-bit instruction are used in TX2. As Figure 4.18 depicts, TitanX in single-precision floating point (FP32), is, on average, 12× faster than TX2. The speedup grows to 19× when 8-bit mode is used. While GPUs can benefit from using as low as 8-bits, Bit Fusion can extract performance benefits for as low as 2-bit operations. Using bit-level composability, Bit Fusion provides a 16× speedup over TX2. The VGG-7 benchmark sees the maximum gains of 30× and 48× performance from Titan Xp and Bit Fusion, respectively. The high degrees of parallelism in VGG-7 enables both Titan Xp and Bit Fusion to utilize all the available on-chip compute resources. Bit Fusion, while consuming 895 milliwatts of power, is only 16% slower than the 250-Watt Titan Xp that uses 8-bit computations, almost matching its performance.

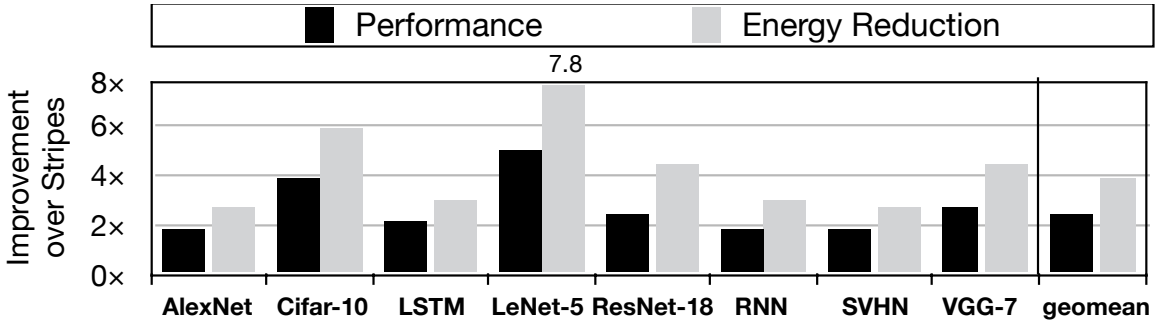


Figure 4.19: Bit Fusion performance and energy improvements over Stripes.

Comparison to Stripes

Performance compared to Stripes. Figure 4.19 presents the performance and energy benefits of Bit Fusion in comparison with Stripes. On average, Bit Fusion provides 2.6× speedup over Stripes. Stripes uses bit-serial computations to support variable bitwidths just for DNN weights. As opposed to Stripes, the Bit Fusion architecture offers dynamically composable BitBricks to support flexible bitwidths for both inputs and weights in DNNs. Bit Fusion achieves the highest speedup of 5.2× and lowest speedup of 1.8× over Stripes for benchmarks LeNet-5 and AlexNet, respectively. ResNet-18 which is the most recent and the biggest of the benchmarks sees 2.6× performance benefits as it can use low bitwidth on both operands. AlexNet uses 8-bit inputs/weights for the first convolution layer and the last fully-connected layer. The two 8-bit layers limit the benefits of Bit Fusion over Stripes. Benchmark LeNet-5, on the other hand, uses low bitwidths for both inputs and weights, resulting in the highest performance benefits with Bit Fusion.

Energy reduction compared to Stripes. Figure 4.19 also depicts the improvement in energy when Bit Fusion is compared to Stripes. As mentioned, Bit Fusion benefits from reduction in both computation and memory access at lower bitwidths for *both* inputs and weights. On average, Bit Fusion reduces energy consumption by 3.9× over Stripes. LeNet-5 sees the highest energy reduction of 7.8×, while benchmark AlexNet sees the least energy reduction of 2.7× over Stripes. For ResNet-18, the energy is reduced by a factor of 4×.

Bit Fusion offers a fundamentally different approach from Stripes and explores the dimension of bit-level dynamic composabililty, which significantly improves performance and energy.

4.4 Related Work

A growing body of related works develop DNN accelerators. Bit Fusion fundamentally differs from prior work as it introduces and explores a new dimension of bit-level composable architectures that can dynamically match the bitwidth required by DNN operations. Bit Fusion aims to minimize both computations and communications in the finest granularity possible without compromising on the DNN accuracy. Below, we discuss the most related work.

Precision flexibility in DNNs. Stripes [89] and Tartan [158] use bit-serial compute units to provide precision flexibility for inputs at the cost of additional area overhead. Both works provide performance and efficiency benefits that are proportional to the precision reduction for inputs. We directly compare the benefits of Bit Fusion to Stripes in Section 4.2.1. UNPU [169] fabricates a bit-serial DNN accelerator at 65 nm, similar to Stripes [89]. Loom [168] uses bit-serial computation for precision flexibility. DeepRecon [178] skips stages of a fully-pipelined floating-point-multiplier to perform either one 16-bit, two 12-bit, or four 8-bit multiplications. In contrast, the Fusion Units are spatial designs that use combinational logic to dynamically compose and decompose 2-bit multipliers (BitBricks) to construct variable bitwidth multiply-add units. Moons et al. propose aggressive voltage scaling techniques at low precision for increased energy efficiency at constant throughput by turning off parts of the multiplier [167, 179]. As such, they do not offer fusion capabilities. TPU [137] proposes a systolic architecture for DNNs and supports 8-bit and 16-bit precision. This work, on the other hand, proposes an architecture that dynamically composes low-bitwidth compute units (BitBricks) to match the bitwidth requirements of DNN layers.

Binary DNN accelerators. Several inspiring works have explored ASIC and FPGA accelerators optimized for Binary DNNs. FINN [180] uses FPGAs for accelerating Binary DNNs, while YodaNN [181] and BRein [182] propose an ASIC accelerator for binary DNNs. Kim, et al. [183] decompose the convolution weights for binary CNNs to improve performance and energy efficiency. The above works focus solely on binary DNNs to achieve high performance at the cost of classification accuracy. Bit Fusion, on the other hand, flexibly matches the bitwidths of DNN operations for performance/energy benefits without losing accuracy.

Sparse Accelerators for DNNs. EIE [69], Cambricon-X [90], Cnvlutin [71], and SCNN [184] explore the sparsity in the DNN layers and use zero-skipping to provide performance and energy-efficiency benefits. Orthogonal to the works above, Bit Fusion explores the dimension of bit-flexible accelerators for DNNs.

Other ASIC accelerators for DNNs. DaDianNao [36] uses eDRAM to eliminate off-chip accesses and provide high performance and efficiency for DNNs. PuDianNao [43] is an accelerator designed for machine learning, but does not support CNNs. Minerva [70] proposes operation pruning and data quantization techniques to reduce power consumption for ASIC acceleration. Eyeriss [46, 47] presents an optimized row-stationary dataflow for DNNs to improve efficiency. Tetris [157] and Neurocube [159] propose 3-D stacked DNN accelerators to provide high bandwidth for DNN operations. ISAAC [163], PipeLayer [165], and Prime [164] use resistive RAM (ReRAM) for accelerating DNNs. Ganax [185] uses a SIMD-MIMD architecture to support DNNs and generative models. Snapea [186] employs early termination to skip computations.

Instruction Sets for DNNs. Cambricon [67] provides an ISA to express the different computations in a DNN using vector and matrix operations without significant loss in efficiency over DaDianNao. DNNWEAVER [24] proposes a coarse grained ISA to express layers of DNNs, which are first translated to micro-codes for FPGA acceleration. Unlike prior work, the Fusion-ISA proposed in the work is designed to enable bit-level flexibility for acceler-

ating DNNs. Further, the Fusion-ISA uses `loop` instructions with iterative semantics to significantly reduce instruction footprint.

Code optimization techniques. Alwani, et. al [92] propose layer-fusion, that combines multiple convolutional layers to save off-chip accesses for FPGA acceleration of CNNs. Escher [187] proposes a CNN FPGA accelerator using flexible buffering that balances the off-chip accesses for inputs and weights in CNNs. The above works have inspired the code-optimizations explored in this chapter, however, the key contribution of this work is a bit-level flexible DNN accelerator.

Software techniques for Binary/XNOR DNNs. QNN [155] shows that efficient GPU kernels for XNOR-based binary DNNs can provide up to 3.4× improvement in performance. XNOR-Net [188] shows that specialized libraries for Binary/XNOR-nets can achieve 58× performance on CPUs. In contrast, Bit Fusion is an ASIC accelerator architecture that supports a wide range of bitwidths (binary to 16-bits) for DNNs with no accuracy loss.

Core Fusion and CLPs. Core Fusion [189] and CLPs [190] are dynamically configurable chip multiprocessors that a group of independent processors can fuse and form a more capable CPU. In contrast to these inspiring works, Bit Fusion performs the composition in the bit level rather than at the level of full-fledged cores.

4.5 Conclusion

Deep neural networks use abundant computation, but can withstand very low bitwidth operations without any loss in accuracy. Leveraging this property of DNNs, we develop Bit Fusion, a bit-level dynamically composable architecture, for their efficient acceleration. The architecture comes with an ISA that enables the software to utilize this bit-level fusion capability to maximize the parallelism in computations and minimize the data transfer in the finest granularity possible. We evaluate the benefits of Bit Fusion by synthesizing the Verilog implementation of the proposed microarchitecture in 45 nm technology node and using cycle accurate simulations with eight real-world DNNs that require different

bitwidths in their layers. Bit Fusion achieves significant speedup and energy benefits compared to state-of-the-art accelerators.

CHAPTER 5
FUTURE DIRECTIONS

Leveraging algorithmic insights is imperative to push the boundaries of hardware acceleration. This thesis is an initial step in providing a unified full-stack solution for the edge-to-cloud continuum that exploits algorithmic properties of deep learning to provide orders of magnitude higher performance/energy efficiency over the conventional general purpose computing stack. Community engagement and contribution are vital for providing a general, efficient, and accessible platform for accelerating Deep Learning. To facilitate such engagement, DNNWEAVER and Bit Fusion have been made publicly available at http://dnnweaver.org and http://act-lab.org/artifacts/bitfusion/.

At the same time, this thesis opens up several avenues for research that aim to provide programmability, performance, and efficiency, discussed below.

Specialized compute stack for emerging technologies. The final chapter of this thesis (Chapter 4) explores leveraging mixed-signal acceleration to gain significant energy/performance benefits over purely-digital designs. The enabling technology that allowed us to replace digital compute units with mixed-signal compute units is the idea that the highly-parallel and highly-regular convolutions/matrix-multiplications in deep learning can be decomposed into binary or low bitwidth operations. ReRAM technologies, including Magneto-Resistive RAM (or MRAM) is a particularly interesting future work since it allows both compute and storage to be performed on the same die, yielding several orders of magnitude increase in performance and efficiency. While the binary/low-bitwidth operations map efficiently to such non-traditional and emerging technologies, they suffer from limited encoding range and the susceptibility to noise is high. As such, one line of research I would like to explore is to develop full-stack solutions that overcome the challenges of emerging technologies through a hardware-software co-design.

Embedding domain-specific accelerators in the network for scale-out acceleration. The highly parallel nature of deep learning presents a research opportunity for scale-out multi-chip acceleration. However, overcoming the latency and energy penalty for data transfer can limit the potential for scale-out acceleration. This challenge is further exacerbated when the accelerators rely on commodity networking infrastructure, wherein the CPU plays an active role in data transfers across a network of accelerator-equipped servers. As such, I would like to explore the integration of accelerations within the network that can both process data and perform data transfers over the network with no participation from the host CPU. An important research challenge here is to develop a hardware-managed in-network caching and coherence protocols that overcome the costly data transfers over the network to provide significant gains in performance and efficiency. The benefits from such an infrastructure hinge upon developing hardware-level protocols that can enable the offloading of compute-intensive operations to accelerators embedded in the network as well as on software abstractions that enable applications to take advantage of these capabilities.

Decoupling algorithm from hardware for deep learning. Regularity and homogeneity in computations is necessary to achieve high performance in traditional architectures like CPUs/GPUs for deep learning. The need for regularity in the algorithms to obtain performance from traditional processor architectures has been one of the major driving forces for evolution of the highly regular and highly parallel modern deep learning algorithms. As a future direction, I wish to explore irregular and heterogeneous-precision neural network topologies that are decoupled from the constraints and limitations imposed by existing processor architectures. Exploiting the re-configurability offered by FPGAs is central to this line of research. FPGAs, as this thesis shows in Chapter 2, can be programmed to be specialized at the hardware level to topologies of individual deep learning models. While prior works, including my own, have looked extensively at the inference phase of deep learning, FPGAs may offer significant efficiency benefits over traditional CPU/GPU architectures for irregular and heterogeneous-precision neural network topologies.

REFERENCES

[1] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh, "Neural acceleration for gpu throughput processors," in *Proceedings of the 48th annual IEEE/ACM international symposium on microarchitecture*, ACM, 2015.

[2] *GeForce 400 series*, http://en.wikipedia.org, 2015. [Online]. Available: http://en.wikipedia.org/wiki/GeForce_400_series.

[3] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-tuning approximation for graphics engines," in *International Symposium on Microarchitecture (MICRO)*, 2013.

[4] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *ASPLOS*, 2014.

[5] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization," ser. International Symposium on Computer Architecture (ISCA), 2014.

[6] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant gpu applications," *Multimedia, IEEE Transactions on*, 2013.

[7] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *International Symposium on Microarchitecture (MICRO)*, 2012.

[8] R. St. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," in *International Symposium on Computer Architecture (ISCA)*, 2014.

[9] B. Grigorian, N. Farahpour, and G. Reinman, "BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[10] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "SNNAP: Approximate computing on programmable socs via neural acceleration," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[11] L. McAfee and K. Olukotun, "EMEURO: A framework for generating multi-purpose accelerators via deep learning," in *CGO*, 2015.

[12] B. Grigorian and G. Reinman, "Accelerating divergent applications on SIMD architectures using neural networks," in *ICCD*, 2014.

[13] (). NVIDIA corporation. NVIDIA CUDA SDK code samples, [Online]. Available: https://developer.nvidia.com/gpu-computing-sdk..

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.

[15] jMonkeyEngine, 2015. [Online]. Available: http://jmonkeyengine.org/.

[16] O. A. Aguilar and J. C. Huegel, "Inverse kinematics solution for robotic manipulators using a cuda-based parallel genetic algorithm," *AAI*, 2011.

[17] M. Creel and M. Zubair, "A high performance implementation of likelihood estimators on gpus," in *CES*, 2013.

[18] A Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *ISPASS*, 2009.

[19] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling energy optimizations in gpgpus," in *ISCA*, 2013.

[20] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.

[21] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *MICRO*, 2007.

[22] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *MICRO*, 2012.

[23] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," *Archit. News*, 2000.

[24] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Misra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *International Symposium on Microarchitecture (MICRO)*, 2016.

[25] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars, "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[26] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *ICASSP*, 2013.

[27] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[28] G. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.

[30] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," 2015.

[31] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun, "Deep image: Scaling up image recognition," *arXiv preprint arXiv:1501.02876*, 2015.

[32] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," 2013.

[33] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *International Symposium on Computer Architecture (ISCA)*, 2011.

[34] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Toward dark silicon in servers," *IEEE Micro*, vol. 31, no. 4, pp. 6–15, 2011.

[35] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[36] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, "Dadiannao: A machine-learning supercomputer," in *International Symposium on Microarchitecture (MICRO)*, 2014.

[37] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2015.

[38] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *CVPRW*, 2011.

[39] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[40] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[41] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *International Symposium on Microarchitecture (MICRO)*, 2011.

[42] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Prashanth, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. R. Larus, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *International Symposium on Computer Architecture (ISCA)*, 2014.

[43] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[44] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *International Symposium on Computer Architecture (ISCA)*, 2015.

[45] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "TABLA: A unified template-based framework for accelerating statistical machine learning," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[46] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[47] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 52, no. 1, pp. 127–138, 2017.

[48] J. Sim, J. S. Park, M. Kim, D. Bae, Y. Choi, and L. S. Kim, "14.6 a 1.42tops/w deep convolutional neural network recognition processor for intelligent ioe systems," in *ISSCC*, 2016.

[49] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 41, 2013, pp. 24–35.

[50] F. Conti and L. Benini, "A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015.

[51] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *CVPRW*, 2014.

[52] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *ISCAS*, 2010.

[53] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[54] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[55] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[56] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *arXiv preprint arXiv:1312.6229*, 2013.

[57] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," *Computer Science Department, University of Toronto, Tech. Rep*, vol. 1, no. 4, p. 7, 2009.

[58] Y. LeCun, C. Cortes, and C. Burges, "MNIST Handwritten Digit Database," *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, vol. 2, 2010.

[59] M. Lin, Q. Chen, and S. Yan, "Network in network," *CoRR*, vol. abs/1312.4400, 2013. [Online]. Available: http://arxiv.org/abs/1312.4400.

[60] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[61] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *IJCV*, vol. 115, no. 3, pp. 211–252, 2015.

[62] S. Cheng and J. Wawrzynek, "High level synthesis with a dataflow architectural template," *arXiv preprint arXiv:1606.06451*, 2016.

[63] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 38, 2010, pp. 247–257.

[64] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

[65] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

[66] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "Deepburning: Automatic generation of fpga-based learning accelerators for the neural network family," in *Design Automation Conference (DAC)*, 2016.

[67] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[68] L. Song, Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li, "C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization," in *Design Automation Conference (DAC)*, 2016.

[69] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[70] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate

deep neural network accelerators," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[71] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[72] *Apache Spark*, 2017. [Online]. Available: https://spark.apache.org/.

[73] *Apache Hadoop*, 2017. [Online]. Available: http://hadoop.apache.org/.

[74] *Snickerdoodle: Affordable FPGA platform for powering everything robots, drones, and computer vision*, 2017. [Online]. Available: http://krtkl.com/.

[75] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *MICRO*, 2016.

[76] *Amazon EC2 F1 instances: Run custom FPGAs in the AWS cloud*, 2017. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/.

[77] X. Feng, A. Kumar, B. Recht, and C. Ré, "Towards a unified architecture for in-rdbms analytics," in *SIGMOD*.

[78] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[79] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "On parallelizability of stochastic gradient descent for speech dnns," in *ICASSP*, 2014.

[80] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *NIPS*, 2010.

[81] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao, "Optimal distributed online prediction using mini-batches," *Journal of Machine Learning Research*, vol. 13, no. Jan, pp. 165–202, 2012.

[82] J. Langford, A. Smola, and M. Zinkevich, "Slow learners are fast," in *NIPS*, 2009.

[83] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. D. Walker, "Efficient large-scale distributed training of conditional maximum entropy models," in *NIPS*, 2009.

[84] D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey, "Distributed deep learning using synchronous stochastic gradient descent," *arXiv:1602.06709 [cs]*, 2016.

[85] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD," in *ICLR Workshop Track*, 2016.

[86] Intel Altera, *Arria 10 architecture*, 2017. [Online]. Available: https://www.altera.com/products/fpga/arria-serqies/arria-10/features.html.

[87] *Spark MLlib: Apache spark's scalable machine learning library*. [Online]. Available: http://spark.apache.org/mllib/.

[88] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *MICRO*, 2016.

[89] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *International Symposium on Microarchitecture (MICRO)*, 2016.

[90] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *International Symposium on Microarchitecture (MICRO)*, 2016.

[91] Y. Ji, Y. Zhang, S. Li, and P. Chi, "NEUTRAMS: Neural network transformation and co-design under neuromorphic hardware constraints," in *MICRO*, 2016.

[92] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerator," in *International Symposium on Microarchitecture (MICRO)*, 2016.

[93] I. Stamoulias and E. S. Manolakos, "Parallel architectures for the knn classifier – design of soft ip cores and fpga implementations," *ACM Transactions on Embedded Computer Systems*, vol. 13, no. 2, 22:1–22:21, Sep. 2013.

[94] E. Manolakos and I. Stamoulias, "Ip-cores design for the knn classifier," in *ISCAS*, 2010.

[95] H. Hussain, K. Benkrid, H. Seker, and A. Erdogan, "Fpga implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data," in *AHS*, 2011.

[96] T. Maruyama, "Real-time k-means clustering for color images on reconfigurable hardware," in *ICPR*, 2006.

[97] A. Filho, A. Frery, C. de Araujo, H. Alice, J. Cerqueira, J. Loureiro, M. de Lima, M. Oliveira, and M. Horta, "Hyperspectral images clustering on reconfigurable hardware using the k-means algorithm," in *SBCCI*, 2003.

[98] M. Papadonikolakis and C. Bouganis, "A heterogeneous fpga architecture for support vector machine training," in *FCCM*, 2010.

[99] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. Graf, "A massively parallel fpga-based coprocessor for support vector machines," in *FCCM*, 2009.

[100] A. Majumdar, S. Cadambi, and S. Chakradhar, "An energy-efficient heterogeneous system for embedded learning and classification," *IEEE Embedded Systems Letters*, vol. 3, no. 1, pp. 42–45, 2011.

[101] A. Majumdar, S. Cadambi, M. Becchi, S. T. Chakradhar, and H. P. Graf, "A massively parallel, energy efficient programmable accelerator for learning and classification," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 1, 6:1–6:30, 2012.

[102] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *CVPRW*, 2011.

[103] A. Roldao and G. A. Constantinides, "A high throughput fpga-based floating point conjugate gradient implementation for dense matrices," *ACM Transactions on Reconfigurable Technology System*, vol. 3, no. 1, Jan. 2010.

[104] G. Morris, V. Prasanna, and R. Anderson, "A hybrid approach for mapping conjugate gradient onto an fpga-augmented reconfigurable supercomputer," in *FCCM*, 2006.

[105] D. DuBois, A. DuBois, T. Boorman, C. Connor, and S. Poole, "An implementation of the conjugate gradient algorithm on fpgas," in *FCCM*, 2008.

[106] D. Kesler, B. Deka, and R. Kumar, "A hardware acceleration technique for gradient descent and conjugate gradient," in *SASP*, 2011.

[107] *A variant of mnist dataset with 8 millions records*. [Online]. Available: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#mnist8m.

[108] J. P. Pinto, "Multilayer perceptron based hierarchical acoustic modeling for automatic speech recognition," PhD thesis, EPFL, 2010.

[109] B. Zhou, "High-frequency data and volatility in foreign-exchange rates," *Journal of Business & Economic Statistics*, vol. 14, no. 1, 2008.

[110] S Dhanya and R. V. Kumari, "Comparison of various texture classification methods using multiresolution analysis and linear regression modelling," *Springerplus*, vol. 5, no. 54, 2016.

[111] M. Segal, K. Dahlquist, and B. Conklin, "Regression approaches for microarray data analysis," *Journal of Computational Biology*, vol. 10, no. 6, 2003.

[112] D Singh, P Febbo, K Ross, D Jackson, J Manola, C Ladd, P Tamayo, A Renshaw, A. A. D, J Richie, E Lander, M Loda, P Kantoff, T Golub, and W Sellers, "Gene expression correlates of clinical prostate cancer behavior," *Cancer Cell*, vol. 1, no. 2, 2002.

[113] I. Cantador, P. Brusilovsky, and T. Kuflik, "Movielens dataset," in *HetRec*, 2011.

[114] Grouplens. (2017). Movielens dataset, [Online]. Available: http://grouplens.org/datasets/movielens/.

[115] *Netflix Prize Data Set*. [Online]. Available: http://www.netflixprize.com/.

[116] J. Weston, S. Mukherjee, O. Chapelle, M. Pontil, T. Poggiott, and V. Vapnik, "Feature selection for svms," in *NIPS*, 2000.

[117] *Spark gpu and simd support*. [Online]. Available: https://github.com/kiszk/spark-gpu.

[118] R. Bordawekar, *Accelerating spark workloads using gpus*. [Online]. Available: https://www.oreilly.com/learning/accelerating-spark-workloads-using-gpus.

[119] *GPUEnabler*. [Online]. Available: https://github.com/ibmsparkgpu/gpuenabler.

[120] *CUDA-MLlib*. [Online]. Available: https://github.com/ibmsparkgpu/cuda-mllib.

[121] A. Athanasopoulos, A. Dimou, V. Mezaris, and I. Kompatsiaris, "GPU acceleration for support vector machines," in *12th International Workshop on Image Analysis for Multimedia Interactive Services*, 2011.

[122] Facebook AI Research, *Caffe2*, https://caffe2.ai/.

[123] *CUDA v8.0*, 2017. [Online]. Available: https://developer.nvidia.com/cuda-toolkit.

[124] *cuDNN v7.0*, 2017. [Online]. Available: https://developer.nvidia.com/cudnn.

[125] *Wattsup .net meter.* 2017.

[126] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," in *ASPLOS*, 2011.

[127] M. Li, T. Zhang, Y. Chen, and A. J. Smola, "Efficient mini-batch training for stochastic optimization," in *KDD*, 2014.

[128] A. Cotter, O. Shamir, N. Srebro, and K. Sridharan, "Better mini-batch algorithms via accelerated gradient methods," in *NIPS*, 2011.

[129] M. T. c, A. Bijral, P. Richtárik, and N. Srebro, "Mini-batch primal and dual methods for svms," in *ICML*, 2013.

[130] O. Dekel, O. Shamir, and L. Xiao, "Optimal distributed online prediction using mini-batches," *Journal of Machine Learning Research*, vol. 13, no. 1, pp. 165–202, 2012.

[131] R. H. Byrd, G. M. Chin, J. Nocedal, and Y. Wu, "Sample size selection in optimization methods for machine learning," *Mathematical Programming*, vol. 134, no. 1, 2012.

[132] *TABLA source code*. [Online]. Available: http://www.act-lab.org/artifacts/tabla/.

[133] C. Farabet, Y. LeCun, K. Kavukcuoglu, E. Culurciello, B. Martini, P. Akselrod, and S. Talay, "Large-scale fpga-based convolutional networks," *Machine Learning on Very Large Data Sets*, 2011.

[134] C. Donninger, A. Kure, and U. Lorenz, "Parallel brutus: The first distributed, fpga accelerated chess program," in *IPDPS*, 2004.

[135] J. P. Walters, X. Meng, V. Chaudhary, T. Oliver, L. Y. Yeow, D. Nathan, B. Schmidt, and J. Landman, "Mpi-hmmer-boost: Distributed fpga acceleration," *Journal of VLSI Signal Processing*, vol. 48, no. 3, pp. 223–238, 2007.

[136] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengil, M. Liu, D. Lo, S. Alkalay, M. Haselman, C. Boehn, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, T. Juhasz, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, S. Reinhardt, A. Sapek, R. Seera, B. Sridharan, L. Woods, P. Yi-Xiao, R. Zhao, and D. Burger, "Accelerating persistent neural networks at datacenter scale," in *HotChips*, 2017.

[137] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *International Symposium on Computer Architecture (ISCA)*, 2017.

[138] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," *arXiv:1603.04467 [cs]*, 2016.

[139] S. Cheng and J. Wawrzynek, "High Level Synthesis with a Dataflow Architectural Template," in *OLAF*, 2016.

[140] E. S. Chung, J. D. Davis, and J. Lee, "LINQits: Big data on little clients," in *ISCA*, 2013.

[141] A. R. Putnam, D. Bennett, E. Dellinger, J. Mason, and P. Sundararajan, "CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures," in *FPGA*, 2008.

[142] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "Fpmr: Mapreduce framework on fpga," in *International Symposium on Field Programmable Gate Arrays (FPGA)*, ser. FPGA '10, Monterey, California, USA: ACM, 2010, pp. 93–102, ISBN: 978-1-60558-911-4. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723129.

[143] J. Fowers, K. Ovtcharov, K. Strauss, E. Chung, and G. Stitt, "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," in *International Symposium on Field-Programmable Custom Computing Machines*, IEEE, 2014. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=217166.

[144] E. S. Chung, J. C. Hoe, and K. Mai, "Coram: An in-fabric memory architecture for fpga-based computing," in *International Symposium on Field Programmable Gate Arrays (FPGA)*, ACM, 2011, pp. 97–106.

[145] M. King, A. Khan, A. Agarwal, O. Arcas, and Arvind, "Generating infrastructure for fpga-accelerated applications," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, 2013, pp. 1–6.

[146] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[147] I. Ouaiss, S. Govindarajan, V. Srinivasan, and R. Vemuri, "An integrated partitioning and synthesis system for dynamically reconfigurable multi-fpga architectures," *Lecture Notes in Computer Science*, vol. 1385-1388, pp. 31–36, 1999.

[148] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, "Programming and runtime support to blaze fpga accelerator deployment at datacenter scale," in *SoCC*, 2016.

[149] Z. Wang, S. Zhang, B. He, and W. Zhang, "Melia: A mapreduce framework on opencl-based fpgas," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3547–3560, 2016.

[150] D. Diamantopoulos and C. Kachris, "High-level synthesizable dataflow mapreduce accelerator for fpga-coupled data centers," in *SAMOS*, 2015.

[151] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI*, 2004.

[152] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *CoRR*, vol. abs/1606.06160, 2016. [Online]. Available: http://arxiv.org/abs/1606.06160.

[153] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *arXiv preprint arXiv:1612.01064*, 2016.

[154] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

[155] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *arXiv preprint arXiv:1609.07061*, 2016.

[156] A. K. Mishra, E. Nurvitadhi, J. J. Cook, and D. Marr, "WRPN: wide reduced-precision networks," *CoRR*, vol. abs/1709.01134, 2017. arXiv: 1709.01134. [Online]. Available: http://arxiv.org/abs/1709.01134.

[157] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[158] A. Delmas, S. Sharify, P. Judd, and A. Moshovos, "Tartan: Accelerating fully-connected and convolutional layers in deep learning networks by exploiting numerical precision variability," *CoRR*, vol. abs/1707.09068, 2017. arXiv: 1707.09068. [Online]. Available: http://arxiv.org/abs/1707.09068.

[159] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[160] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *CVPRW*, 2014.

[161] J. Sim, J. S. Park, M. Kim, D. Bae, Y. Choi, and L. S. Kim, "14.6 a 1.42tops/w deep convolutional neural network recognition processor for intelligent ioe systems," in *ISSCC*, 2016.

[162] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *International Symposium on Microarchitecture (MICRO)*, 2016.

[163] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[164] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[165] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[166] *Apple a11-bionic neural engine*, https://en.wikipedia.org/wiki/Apple_A11.

[167] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi," in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2017.

[168] S. Sharify, A. D. Lascorz, P. Judd, and A. Moshovos, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," *CoRR*, vol. abs/1706.07853, 2017. arXiv: 1706.07853. [Online]. Available: http://arxiv.org/abs/1706.07853.

[169] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "Unpu: A 50.6 tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision," in *ISSCC*, 2018.

[170] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[171] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *NIPS workshop on deep learning and unsupervised feature learning*, vol. 2011, 2011, p. 5.

[172] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[173] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.

[174] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[175] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of english: The penn treebank," *Computational linguistics*, vol. 19, no. 2, pp. 313–330, 1993.

[176] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques," in *ICCAD*, 2011.

[177] *Nvidia tensor rt 4.0*, https://developer.nvidia.com/tensorrt.

[178] T. Rzayev, S. Moradi, D. H. Albonesi, and R. Manohar, "Deeprecon: Dynamically reconfigurable architecture for accelerating deep neural networks," *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 116–124, 2017.

[179] B. Moons and M. Verhelst, "A 0.3–2.6 tops/w precision-scalable processor for real-time large-scale convnets," in *VLSI-Circuits*, 2016.

[180] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[181] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An ultra-low power convolutional neural network accelerator based on binary weights," *CoRR*, vol. abs/1606.05487, 2016. arXiv: 1606.05487. [Online]. Available: http://arxiv.org/abs/1606.05487.

[182] K. Ando, K. Ueyoshi, K. Orimo, H. Yonekawa, S. Sato, H. Nakahara, M. Ikebe, T. Asai, S. Takamaeda-Yamazaki, T. Kuroda, *et al.*, "Brein memory: A 13-layer 4.2 k neuron/0.8 m synapse binary/ternary reconfigurable in-memory deep neural network accelerator in 65 nm cmos," in *Symposium on VLSI Circuits*, 2017.

[183] H. Kim, J. Sim, Y. Choi, and L.-S. Kim, "A kernel decomposition architecture for binary-weight convolutional neural networks," in *Design Automation Conference (DAC)*, 2017.

[184] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks," in *International Symposium on Computer Architecture (ISCA)*, 2017.

[185] A. Yazdanbakhsh, H. Falahati, P. J. Wolfe, K. Samadi, H. Esmaeilzadeh, and N. S. Kim, "GANAX: A Unified SIMD-MIMD Acceleration for Generative Adversarial Network," in *International Symposium on Computer Architecture (ISCA)*, 2018.

[186] V. Aklaghi, A. Yazdanbakhsh, K. Samadi, H. Esmaeilzadeh, and R. K. Gupta, "Snapea: Predictive early activation for reducing computation in deep convolutional neural networks," in *International Symposium on Computer Architecture (ISCA)*, 2018.

[187] Y. Shen, M. Ferdman, and P. Milder, "Escher: A cnn accelerator with flexible buffering to minimize off-chip transfer," in *FCCM*, 2017.

[188] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," *CoRR*, vol. abs/1603.05279, 2016. arXiv: 1603.05279. [Online]. Available: http://arxiv.org/abs/1603.05279.

[189] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core fusion: Accommodating software diversity in chip multiprocessors," in *International Symposium on Computer Architecture (ISCA)*, 2007.

[190] C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, "Composable lightweight processors," in *International Symposium on Microarchitecture (MICRO)*, 2007.

VITA

Hardik Sharma was born in Ahmedabad, India and raised in Jaipur, India. He received his Bachelors in 2013 from the Indian Institute of Technology, Guwahati. Subsequently, he earned his Masters in 2015 from the department of Electrical and Computer Engineering (ECE) at Georgia Institute of Technology (GaTech). He earned his PhD from the School of Electrical and Computer Engineering at the Georgia Institute of Technology, where he was advised by Prof. Hadi Esmaeilzadeh as a part of the Alternate Computing Technologies (ACT) lab.

Hardik's interests lie in the intersection of Computer Architecture and Machine Learning. Specifically, Hardik is interested in designing specialized compute stacks for accelerating data analytics, statistical machine learning, and deep learning. To maximize the benefits from acceleration, his work leverages algorithmic insights to enable specializations across the compute stack. Hardik has led several open-source projects based on his research that have gained attention from both academia and industry. In recognition of his research, Hardik won the Qualcomm Innovation Fellowship (QInF) in 2018 and won the distinguished paper award at HPCA 2016.

This dissertation was typeset in LaTeX.